VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PŘEKLADAČ DOPOČTOVÝCH ROVNIC ENERGETICKÉHO PRŮMYSLU

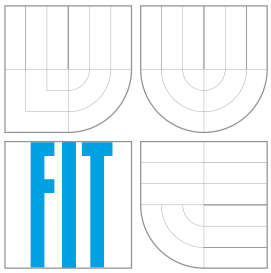
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PETR PEHAL

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PŘEKLADAČ DOPOČTOVÝCH ROVNIC ENERGETICKÉHO PRŮMYSLU

COMPILER OF EQUATION IN ENERGETIC INDUSTRY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — PETR PEHAL
AUTHOR

VEDOUCÍ PRÁCE — Ing. MARTIN ČERMÁK
SUPERVISOR

BRNO 2010

Abstrakt

Bakalářská práce se zabývá specializovaným programovacím jazykem využívaným pro specifické výpočty v systému pro řízení energetických sítí. Nejdříve uvádí obecný význam těchto výpočtů v energetickém průmyslu a následně konkrétní využití jazyka. Představuje syntaxi a sémantiku jazyka společně s jeho zajímavými vlastnostmi. Dále se zaměřuje na návrh a implementaci optimalizačních úprav a požadovaných rozšíření.

Abstract

The bachelor thesis deals with the specialized programming language used for specific calculations in power grid controlling system. First, it presents general significance of these calculations in energetic industry and then particular utilization of the language. It introduces language's syntax and semantics along with its interesting features. Further it focuses on design and implementation of language optimization and extensions.

Klíčová slova

Energetický průmysl, systém RIS, dopočty, dopočtový jazyk, databázový engine, překladač, interpret

Keywords

Energetic industry, system RIS, energetic calculations, language for energetic calculations, database engine, compiler, interpreter

Citace

Petr Pehal: Překladač dopočtových rovnic energetického průmyslu, bakalářská práce, Brno, FIT VUT v Brně, 2010

Překladač dopočtových rovnic energetického průmyslu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Martina Čermáka.

.....
Petr Pehal
19. května 2010

Poděkování

Děkuji vedoucímu mé práce panu Ing. Martinu Čermákovi a pracovníkům z firmy Elektrosystem, a.s. za pomoc při vypracování této bakalářské práce.

Obsah

Kapitola 1

Úvod

Energetický průmysl patří mezi nejvýznamnější odvětví světového průmyslu. Firma Elektrosystem, a.s. vyvíjí již přes 20 let řídící a informační systémy pro energetické společnosti na území České a Slovenské republiky. V současnosti expanduje firma Elektrosystem se svým systémem *RIS* (viz. [2]) do dalších evropských zemí a převádí jej z real-time operačního systému QNX na multiplatformní. S tímto převodem je spojena i konstrukce stávajícího jazyka využívaného pro definování uživatelských dopočtových rovnic.

Cílem této práce je zmodernizovat a optimalizovat stávající implementaci jazyka využívaného v systému *RIS* pro dopočty energetických veličin. Součástí této modernizace je i rozšíření o nové prvky, které uživatelé jazyka požadují.

Druhá kapitola bude věnována stručnému teoretickému úvodu do problematiky překladačů a formálních jazyků. Představí obecnou strukturu překladače a metody využívané při syntaktické analýze.

Třetí kapitola obecně vysvětlí, co dopočty v energetickým průmyslu znamenají a k čemu se využívají. Následně podá krátký úvod do systému *RIS* a představí způsob, jakým jsou dopočty v tomto systému řešeny. Závěr kapitoly bude vyhrazen dvěma základním komponentám systému - komunikačnímu a tabulkovému rozhraní.

Čtvrtá kapitola se bude zabývat již konkrétně dopočtovým jazykem. Ukáže jeho vyjadřovací schopnosti, syntaxi a sémantiku. Popíše specifické vlastnosti, které jazyk odlišují od ostatních známých jazyků. Na konec vysvětlí důvody, které donutily firmu Elektrosystem jazyk modernizovat.

V páté kapitole budou diskutovány požadavky uživatelů na rozšíření. Zároveň zde budou specifikovány případné problémy, který splnění těchto požadavků budou stát v cestě. Dále budou jednotlivé požadavky rozebrány a bude navrženo jejich začlenění do jazyka s přihlédnutím ke specifikovaným omezením.

Vlastní návrh a implementace nového překladače budou popsány v kapitole šesté. Jedná se o první kapitolu, která bude obsahovat vlastní práci autora. Kapitola bude pojednávat o rozdělení projektu na celky a konkrétním způsobu řešení jednotlivých částí.

V závěrečných dvou kapitolách bude diskutován budoucí vývoj překladače a shrnuty dosažené výsledky.

Kapitola 2

Úvod do teorie překladačů

Tato úvodní kapitola poskytuje teoretický základ vhodný pro tvorbu překladačů a krátce představuje metody používané při syntaktické analýze.

2.1 Základní definice a pojmy

Předtím než představíme strukturu překladačů a budeme se věnovat metodám syntaktické analýzy, je vhodné vysvětlit základní pojmy z oblasti formálních jazyků. Tento úvod do formálních jazyků čerpá informace z knihy Automata and Languages od profesora Meduny (viz. [11]).

Na začátek je třeba definovat pojmy *abeceda* a *řetězec*.

Definice 2.1.1. *Abeceda* je konečná, neprázdná množina elementů, které se nazývají *symboly*.

Definice 2.1.2. Nechť Σ je abeceda.

1. ε je řetězec nad abecedou Σ.
2. Pokud x je řetězec nad abecedou Σ a $a \in \Sigma$, potom xa je řetězec nad abecedou Σ.

S využitím těchto dvou pojmů můžeme definovat *jazyk*.

Definice 2.1.3. Nechť Σ je abeceda a nechť Σ^* značí množinu všech řetězců nad Σ. Každá podmnožina $L \subset \Sigma^*$ je *jazyk* nad Σ.

Jazyky rozdělujeme podle počtů řetězců na *konečné* a *nekonečné*.

Definice 2.1.4. Jazyk L je *konečný*, pokud L obsahuje konečný počet řetězců, jinak je jazyk *nekonečný*.

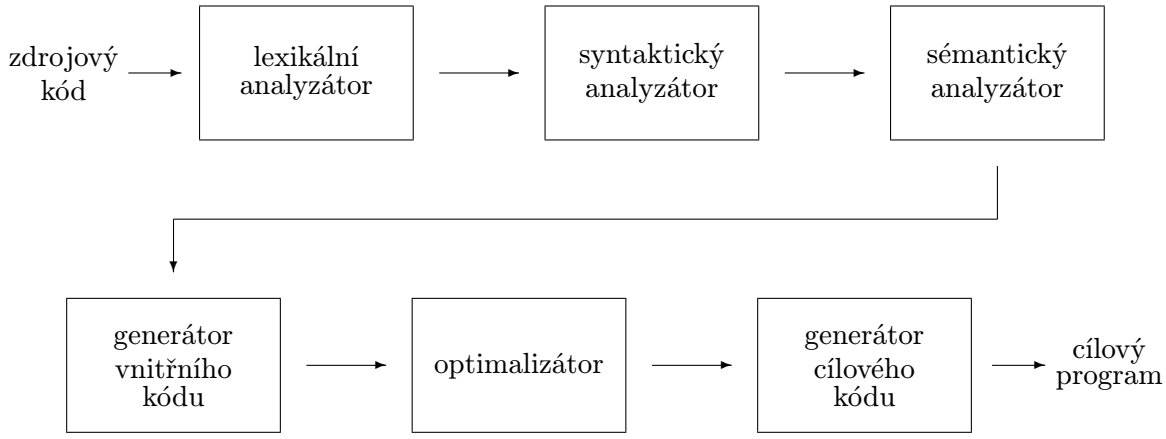
Konečné jazyky je možné specifikovat pomocí výčtu všech prvků. Ovšem jazyky nekonečné, mezi které patří i jazyky programovací, tímto způsobem popsat nelze. Pro jejich popis se využívá syntaktický grafů či Backus-Naurovi formy. Formálně lze popsat nekonečný jazyk s využitím bezkontextové gramatiky.

Definice 2.1.5. Bezkontextová gramatika je čtveřice $G = (N, T, P, S)$, kde

- N je abeceda *neterminálů*
- T je abeceda *terminálů*, přičemž $N \cap T = \varnothing$
- P je konečná množina *pravidel* tvaru $A \to x$, kde $A \in N, x \in (N \cup T)^*$
- $S \in N$ je *počáteční symbol*

2.2 Obecná struktura překladače

Nyní si ji můžeme představit obecnou strukturu překladače. Informace prezentované v této i v následující kapitole jsou převzaty z [9]. Cílem překladače je zpracovat *zdrojový kód* a vytvořit z něj *cílový program*. Překladač při plnění tohoto úkolu postupuje ve fázích, které na sebe navazují. Jednotlivé fáze je možné rozdělit na dvě skupiny: *analýzu* a *syntézu*. Fáze patřící do analýzy vytvářejí ze zdrojového kódu kód přechodný, fáze syntézy tento kód transformují na cílový program. Fáze jsou uspořádány na následujícím obrázku.



Úvodní fází je *lexikální analýza*. V této fázi je zdrojový program transformován na lexikální jednotky zvané *lexémy*. Lexémy jsou reprezentovány *tokeny*.

Druhou fází je *syntaktická analýza*. Syntaktický analyzátor vytváří z řetězce tokenů stromovou strukturu zvanou *derivační strom*. Vytváření toho stromu probíhá podle gramatických pravidel. Podrobněji se bude syntaktické analýze věnovat následující podkapitola.

Sémantická analýza má za úkol kontrolovat sémantiku programu. Jsou zde prováděny akce jako typová kontrola, implicitní konverze či kontrola deklarace proměnných.

Generátor vnitřního kódu produkuje vnitřní reprezentaci programu zvanou *vnitřní kód*. Vnitřní kód je snadno generovatelný a lehce transformovatelný na kód cílový. Příkladem může být tří-adresný kód (jedna instrukce obsahuje 3 operandy).

Úkolem optimalizátoru je upravit vnitřní kód do co možná nejefektivnější formy. Například provádí operace, které je možné vyhodnotit již v době překladu.

Poslední fází je generace cílového kódu. Zde je optimalizovaný vnitřní kód převeden na *cílový program*.

2.3 Metody přístupu k syntaktické analýze

Obecně lze metody rozdělit na tři skupiny: univerzální, pracující shora dolů (top-down) a pracující zdola nahoru (bottom-up). Univerzální metody jsou pro tvorbu překladačů příliš neefektivní a tak nebudou dále diskutovány. Jak napovídají názvy zbylých dvou skupin, metody reprezentující tyto skupiny se liší postupem při vytváření derivačního stromu. Metody pracující shora dolů vytváří derivační strom ze startovacího neterminálu a snaží se pokrýt vstupní řetězec. Zástupci této metody jsou *rekurzivní sestup* a *prediktivní syntaktická analýza*. Metody pracující zdola nahoru postupují ze vstupního řetězce směrem ke startovacímu neterminálu. Do skupiny pracující metodou zdola nahoru patří *precedenční syntaktická analýza* a *LR syntaktická analýza*. Vzhledem k limitovanému rozsahu práce a faktu, že cílem práce je především jazyk dopočtových rovnic, se zde nebudeme metodami podrobněji zabývat. Podrobný popis funkčnosti jednotlivých syntaktických analyzátorů lze nalézt v citované literatuře (viz [9]).

Kapitola 3

Energetický průmysl a dopočty

Úkolem této kapitoly je seznámit čtenáře se smyslem dopočtů v energetickém průmyslu obecně a následně konkretizovat jejich využití v systému *RIS*.

3.1 Využití dopočtů

Řízení energetických sítí je velmi komplexní záležitostí. Informace o stavu energetických veličin jsou získávány z různých zdrojů. *Zdroj* je podle dokumentace společnosti Elektrosystem (viz. [12]) definován jako zařízení nebo program, dodávající do systému informace, které pomáhají k udržení přehledu o energetické síti. Zdrojem mohou být například matematické modely sítě počítající veličiny na základě topologie sítě nebo telemetrie, kde jsou veličiny přímo měřeny. Neméně důležitým zdrojem jsou i dopočty, kde se pomocí veličin známých zjišťují veličiny aktuálně nedostupné či odvozené. Jedná se o obecné dopočty využívané k mnoha různým účelům od základních operací, jako průměrování veličin, až po složité diferenciální výpočty.

Základním rysem dopočtů je, že jejich výpočet není možné řešit obecným algoritmem, který by pracoval nezávisle na konkrétní energetické síti. Pro každou energetickou síť, kterou by daný řídící systém měl spravovat, by bylo třeba výpočty zásadním způsobem modifikovat. Požadovaný výpočet se navíc průběžně doplňuje či mění, aby reflektoval změny v provozu energetických sítí. Proto musí být dopočty definovány uživatelem konkrétního řídícího systému. Je tedy nutné za tímto účelem používat specializovaný programovací jazyk. Případnou možnou alternativou by bylo využít jazyk již existující, který by syntakticky definici uživatelských rovnic umožňoval, a dopočty implementovat jako nadstavbu, například v podobě knihovny.

3.2 Dopočty v systému *RIS*

Následující podkapitoly stručně představují části systému *RIS*, které nějakým způsobem souvisejí s dopočty, případně jsou nutné pro jejich lepší pochopení. Na začátek je vhodné poznamenat, že celý systém *RIS* je implementován v jazyce C. Pouze v současnosti vyvíjené nové multiplatformní uživatelské rozhraní využívá jazyka C++.

3.2.1 Dopočty

Systém *RIS* využívá pro dopočty specializovaný jazyk pro definici rovnic. Zdrojové texty uživatelských dopočtů jsou již velmi rozsáhlé a v současnosti řeší v systému širokou škálu úloh od základních až po velmi závažné. Slouží například k dopočtu odvozených veličin jako jsou sumy a nulové bilance výkonu v uzlech sítě, či napětí na sběrnicích. K nejzávažnějším úkolům dopočtů v systému patří řízení regulace salda České Republiky. Pro ilustraci, zdrojový text dopočtů pro regulátor salda obsahuje zhruba 17 tisíc dopočtových rovnic.

3.2.2 Práce s energetickými veličinami

V systému *RIS* jsou energetické veličiny rozdělovány na dvě základní skupiny – *signály* a *analogy*. Dále jsou veličiny podle podobných vlastností děleny do tříd, ty už ale nejsou z pohledu dopočtů příliš podstatné. Za signály jsou považovány veličiny stavové, například daným uzlem proud protéká, neprotéká, uzel má poruchu. Hodnoty signálů jsou uloženy na několika bitech podle počtu stavů. Valná většina běžných signálů nabývá čtyř stavů (porucha, vypnuto, zapnuto a mezipoloha), pro jejich uložení tedy stačí bity dva. Analogy jsou veličiny, které mají vlastní hodnotu, například proud na daném uzlu. Hodnoty analogů jsou čísla s pohyblivou řádovou čárkou s přesností na 15 desetinných míst.

Obě základní skupiny mají kromě hodnoty samozřejmě celou řadu dalších vlastností, které veličiny blíže specifikují. Mezi nejvýznamnější patří následující tři:

- **kvalita**
  Symbolická konstanta, která má za úkol specifikovat míru správnosti hodnoty veličiny. Většinou je název kvality spojen se zdrojem, ze kterého byla hodnota veličiny zjištěna, například kvalita telemetrická (hodnota naměřená z telemetrií), kvalita dopočtená (hodnota dopočtená v uživatelských dopočtech). Přestože druhů kvalit je celá řada, nejdůležitější je rozlišovat dva případy: zda je kvalita aktuální či neaktuální. Aktuální kvalita značí, že hodnota dané veličiny je s větší či menší mírou přesnosti dodávána do systému a lze ji v daném okamžiku považovat za správnou. Míra přesnosti záleží na konkrétním druhu kvality. Neaktuální kvalita značí, že hodnota veličiny už není anebo ani nikdy nebyla do systému dodávána, případně, že při zjištění hodnoty veličiny nastala chyba. Konkrétně jsou to kvality chybná, nedefinovaná, neexistující a neplatná.

- **alarm**
  Sada příznaků, které určují speciální stavy veličiny v systému, společně se stupněm závažnosti daného stavu. Změna některého z alarmových stavů je ihned vizuálně i akusticky hlášena uživateli pomocí grafického rozhraní systému. Například je to příznak dálkového ovládání veličiny, klidového módu nebo zda je alarm v danou chvíli aktivní.

- **čas**
  Časový údaj, kdy byla naposledy hodnota dodána ze zdroje do systému. V jisté míře ovlivňuje kvalitu. U některých zdrojů není možné zjistit, že přestaly hodnotu veličiny do systému dodávat jiným způsobem, než ověřením platnosti časové známky posledního dodání. Doba platnosti se pro různé zdroje liší.

Tyto tři vlastnosti společně s hodnotou veličiny jsou základem pro dopočty.

3.2.3 Databáze reálného času

Jedním z nejdůležitějších procesů celého systému je databáze reálného času, nazývaná též *Rdb*. Jejím základním úkolem je správa veličin a jejich změn. *Rdb* přijímá informace o energetických veličinách ze sítě, sesbírané informace vyhodnocuje a provádí požadované akce. Následně informuje ostatní programy o změnách, které nastaly, případně které mají samy provést. Komunikuje s programy řídícími statistiky a protokolování. Právě v *Rdb* probíhá většina dopočtů. Někdy jsou ale uživatelské dopočty využívány i jinde, například ve statistikách.

3.2.4 Komunikační interface

Tato i následující podkapitola nesouvisí s dopočty přímo. Jedná se ovšem o dva základní pilíře, na kterých je celý systém *RIS* postaven. Jsou využívány téměř v každé aplikaci systému a nejinak je tomu i u dopočtů.

Prvním základním stavebním kamenem je komunikační interface nazývaný *cmi*. Jedná se o rozhraní určené pro meziprocesovou komunikaci v rámci jednoho stroje i v rámci sítě. Jde o komunikaci typu klient-server, která probíhá prostřednictvím zasílání zpráv. Server se zaregistruje pod daným jménem. Platnost jména může být lokální (pouze na jednom stroji), či globální (v celé síti). Klient se k tomuto jménu musí nejdříve registrovat k odběru zpráv a poskytnout callbackovou funkci, která bude v případě příchodu zpráv volána. Následně si procesy mohou vzájemně vyměňovat zprávy a společně s nimi data.

Interface obsahuje mnoho typů zpráv. V případě potřeby je možné vytvořit typy vlastní. Některé zprávy mají předdefinované akce, ale většinou je interpretace jejich významu ponechána na konkrétní aplikaci. Jako parametry registrovaného callbacku jsou předány struktura popisující odesílatele, struktura popisující zprávu a případná data.

Komunikace není omezena pouze na dva procesy typu klient-server. Záleží pouze na tom, k jakým serverům se proces zaregistruje k odběru zpráv, samozřejmě se k jednomu serveru může registrovat procesů více. Jeden proces může být klientem jiných serverů a zároveň serverem pro další klienty.

Samotný síťový přenos je realizován v režii protokolu TCP/IP, který zaručuje spolehlivé doručení dat. *Cmi* poskytuje i rozhraní pro práci s časovači nebo hardwarovými přerušeními.

3.2.5 Tabulkový interface

Tabulkový interface, *dbi*, je druhým klíčovým prvkem v systému *RIS*. Jde o knihovnu umožňující programům pracovat s tabulkami. Tyto tabulky jsou v systému používány k různým účelům, mezi jinými i jako tabulky databázové. Jedná se o tabulky paměťové, které je možné ukládat na disk a následně načítat. Každá tabulka leží celá v operační paměti procesu, který ji vytvořil nebo načetl. Ukládání může na žádost probíhat asynchronně po nastavených intervalech. Načítat lze i tabulky, které vytvořily a následně uložily jiné procesy.

Tabulka je zpřístupněna přes takzvaný *kurzor*. Ve chvíli vzniku tabulky, ať už vytvořením nové paměťové tabulky nebo načtením dříve uložené, obdrží programátor kurzor, pomocí kterého jsou nad tabulkou prováděny všechny operace. Řádky tabulky jsou pak s využitím kurzoru v programu přístupné jako strukturované datové typy jazyka C. *Dbi* umožňuje tímto způsobem přímo přistupovat k tabulkovým datům. Tento způsob přístupu umožňuje data i modifikovat, čímž lze dosáhnout vysoké efektivity. Při přímé modifikaci je ale programátor plně zodpovědný za konzistenci svých tabulek. Modifikace je samozřejmě možná

i s využitím modifikačních funkcí, které konzistenci za cenu lehké ztráty výkonnosti kontrolují samy.

Nad tabulkou jsou k dispozici následující operace: asociativní vyhledávání, třídění a filtrování. Asociativně vyhledávat, třídit a filtrovat je možné pouze podle sloupců, nad kterými je definován *index*. Na jednu tabulku je možné vytvořit maximálně 16 libovolně složených indexů, které mohou být i nejednoznačné. Pro modifikaci indexových sloupců je nutné použít modifikační funkce. Pokud programátor použije přímou modifikaci pro sloupec, nad kterým je vytvořený index, zničí vnitřní indexovaní, a tím i celou tabulku.

Nejrychlejší přístup ke konkrétním prvkům umožňuje tabulka pomocí vnitřních indexů zvaných *iid*. Jde o celé číslo bez znaménka, které jednoznačně identifikuje každý záznam. K těmto interním indexům z pochopitelných důvodů nelze přistupovat přímo a nejsou ani součástí struktury představující řádky.

Tabulky podporují následující databázové typy: celá a reálná čísla bez a se znaménkem, řetězce fixní a proměnné délky, binární data fixní a proměnné délky, časové údaje a kurzory pro uložení vnořených tabulek. Všechny tyto databázové typy mají svůj ekvivalent v podobě datového typu v jazyce C. Výsledný strukturovaný datový typ reprezentující řádek tabulky je z těchto typů tvořen.

Tabulce je možné registrovat callbackovou funkci, která je volána vždy při provedení dané akce nad tabulkou. Je to vlastně ekvivalent triggerů u SQL databází. Při použití funkcí nad tabulkou je callbacková funkce je volána synchronně v rámci použité akce. Tedy například při modifikaci je callback volán ještě před tím, než program opustí tělo modifikační funkce. Při použití přímé modifikace registrované funkce volány nejsou.

Kromě vytvoření nové tabulky (eventuálně načtení existující z disku) je možné zpřístupnit tabulku, která již je v operační paměti vytvořená. Zpřístupnění vrátí nový kurzor. Jedna tabulka tedy může mít více kurzorů, pomocí kterých se s ní pracuje. Tento způsob násobného přístupu k jedné tabulce se v rámci jednoho programu příliš nepoužívá. Rozumným způsobem využití může být zpřístupnění pouze specifických sloupců tabulky, což tato metoda umožňuje. Zpřístupněné sloupce jsou vázaný na daný kurzor a řádky v podobě struktur následně obsahují pouze specifikované sloupce. Zpřístupnění již otevřené tabulky má ale zásadní význam při meziprocesové komunikaci. *Dbi* totiž využívá služeb *cmi*, a tak je možné vzdáleně zpřístupnit tabulky procesů, které jsou registrovány jako *cmi* servery. Vzdálená komunikace probíhá plně v režii *dbi*. Programátor tedy volá stejné funkce pro práci s tabulkami lokálními i s tabulkami jiných procesů. Přímá modifikace v tomto případě samozřejmě není možná, protože zpřístupněná struktura je pouze kopií originálních dat.

Pro snadnou definici tabulek slouží nástroj zvaný *drcc* (Database ResourCe Compiler). Jde o textový preprocesor, který umožní jako součást zdrojového kódu psát definice tabulek v uživatelsky příjemném tvaru. Z této definice poté vytvoří prototypy struktur, které se dále v kódu používají pro práci s tabulkou. Nejlépe vše vysvětlí příklad v příloze A.

Poznámka: Komunikační i tabulkový interface vznikly koncem 80 let minulého století. *Dbi* bylo původně navrženo pouze pro práci s tabulkami paměťovými. Avšak díky nedostačujícím a nákladným implementacím SQL databází v tomto období bylo upraveno i pro správu dat perzistentních. V současnosti je částečně využívána jako perzistentní datová základna systému databáze Oracle (viz. [3]). I přes výhody, které SQL databáze přináší, valná většina aplikací stále využívá *dbi* a to z důvodu velmi náročnému převodu a také vysoké rychlosti, kterou paměťové tabulky poskytují.

Kapitola 4

Dopočtový jazyk

Kapitola představuje dopočtový jazyk, jeho syntaxi a sémantiku a dva různé typy využívaných dopočtů. Vysvětluje neobvyklý přístup interpretace pomocí volání funkcí rozhraní interpretu z jiných programů a nestandardní chyby, které mohou při interpretaci nastat. Na závěr je shrnut stav dopočtového jazyka a nedostatky, které vedly k rekonstrukci překladače a interpretu. Příloha B obsahuje krátkou ukázku zdrojového kódu jazyka. Nyní doporučujeme do přílohy pouze krátce nahlédnout pro získání obrazné představy o podobě jazyka.

4.1 Koncepce jazyka

Dopočtový jazyk je jazykem interpretovaným. Koncepce práce s jazykem je následující. Zdrojový kód je nejdříve přeložen do binárního kódu. Tato operace je na samotné interpretaci zcela nezávislá a provádí se většinou ihned po dokončení dopočtového souboru.

Interpret je implementován pouze jako knihovna. Poskytuje API umožňující ostatním programům pracovat s přeloženými dopočtovými soubory a interpretovat je. S využitím rozhraní interpretu je binární soubor načten do paměti programu, který s dopočty pracuje. Tento krok je označován jako zavedení nebo instalace dopočtu. Výpočet následně probíhá podle typu dopočtu, dokud není dopočtový soubor pomocí rozhraní interpretu z programu odinstalován.

4.2 Typy dopočtů

Jazyk umožňuje definovat dopočty dvojího typu: dopočty periodické a dopočty změnové. Ze syntaktického a sémantického hlediska není mezi dopočtovými rovnicemi obou typů žádný rozdíl. Zásadní rozdíl nastává až při jejich provádění.

4.2.1 Dopočty změnové

Rovnice jsou po zavedení provedeny jednou jako celek v pořadí, v jakém byly definovány. Následně knihovna interpretu sleduje změny veličin, které do rovnic vstupují. Přepočet rovnic nastává pouze v případě změny některé ze sledovaných veličin, přičemž jsou interpretovány pouze rovnice, které danou veličinu obsahují jako vstupní parametr. Tento výpočet může spustit další změnový dopočet, pokud jsou výstupní parametry z právě počítaných rovnic vstupními parametry rovnic jiných. Interpret přitom musí kontrolovat cyklické závislosti rovnic, aby změnový dopočet nikdy neskončil v nekonečné smyčce. Po zavedení binárního souboru do programu je tedy výpočet již plně v režii interpretu.

4.2.2 Dopočty periodické

Rovnice jsou po zavedení opakovaně prováděny vždy po uplynutí dané periody jako celek. Pořadí provádění rovnic vždy odpovídá uspořádání rovnic ve zdrojovém souboru. Implicitní perioda výpočtu je 5 sekund. Periodu lze nastavit i programově na zadaný počet sekund.

V případě periodických dopočtů ale sledování času a spuštění dopočtu po uplynutí dané periody interpret neprovádí. Sledovat čas a opakovaně volat funkci pro spuštění dopočtu musí sám program, do kterého byl binární soubor zaveden. Zároveň je nutné vždy předat informaci o předchozím a aktuálním čase dopočtu.

Toto chování se může zdát zvláštní, ale má své zdůvodnění. Periodické dopočty se většinou počítají z periodicky telemetricky měřených veličin. Hodnoty periodicky měřených veličin přicházejí do systému v takzvaných *řezech* (viz. [12]). *Řez* je časový úsek, po který je považována hodnota periodicky měřené veličiny za neměnnou. Telemetrické měření se ale může například vlivem síťového provozu opozdit a upravení hodnoty pro daný řez v databázi až po přečtení hodnoty interpretem by způsobilo chybné výsledky dopočtu. Volání funkce pro periodické dopočty ale není v režii interpretu, a proto je možné jej o chvíli pozdržet a vyčkat na výsledky telemetrického měření. V periodických dopočtech se však užívá časově závislých funkcí – integrálů a průměrů. Při opožděném volání by byl výsledek zkreslen tím, že funkce není volána ve správném časovém okamžiku. Funkci pro periodický dopočet se tedy předá čas očekávaný podle délky periody i přesto, že perioda nebyla striktně dodržena. Tohoto triku využívá *Rdb*. Ve chvíli přechodu na nový *řez* se snaží počkat s voláním periodické dopočtové funkce, dokud nebude mít informace ze všech potřebných telemetrických měření. Poté dopočtové funkci „zalže“ o aktuálním čase a zamezí tím chybám v dopočtech.

4.3 Syntaxe a sémantika jazyka

Jazyk dopočtových rovnic je ze syntaktického hlediska inspirován jazykem C. Jedná se o rovnice ve tvaru

$$a = b;$$

kde a je proměnná (výstupní parametr) a b je libovolný výraz obsahující vstupní parametry. Vícenásobné přiřazení výstupní hodnoty proměnné není v jedné rovnici možné.

4.3.1 Vestavěné funkce

Jazyk poskytuje množství vestavěných funkcí. Jsou zde funkce matematické, jako absolutní hodnota, mocnina, či sinus, známé ze standardní matematické knihovny jazyka C (viz. [4]). Početnější skupinou jsou ale funkce specializované na energetiku. Jednak jsou to funkce rázu obecnějšího, které najdou využití bez ohledu na to, zda se pracuje se *signály* či *analogy* nebo zda probíhá dopočet periodický či změnový. Příkladem může být funkce *alr_rc*, která využívá příznaků z alarmu a testuje, zda je veličina dálkově ovládaná. Dále v jazyce najdeme již zmíněné funkce specifické pro dopočty periodické, při jejichž výpočtu se využívá čas aktuálního i předchozího dopočtu. Jsou to výpočty konečných a okamžitých integrálů a průměrů. Specializace funkcí jde ale ještě dál. Najdeme zde například velmi složitou funkci *split*, která byla navržena speciálně pro dopočty regulace salda a jinde nenajde uplatnění.

Některé vestavěné funkce umějí zpracovávat proměnný počet argumentů. Jsou to pomocné funkce *min* a *max*, ale především složité funkce jako výše zmiňovaný *split* nebo *otrafo*.

Podle specifikace syntaxe jazyka není možné získat více než jednu výstupní hodnotu z jedné rovnice. To je ale v případě některých vestavěných funkcí velmi omezující. Proto byl zaveden rozšiřující formát rovnice ve tvaru:

$$f(in_1, \ldots, in_n :: out_1, \ldots, out_n);$$

kde f je vestavěná funkce vyžadující více výstupních parametrů, $in_1, \ldots, in_n$ jsou libovolné výrazy (vstupní parametry) a $out_1, \ldots, out_n$ jsou proměnné (výstupní parametry). Tento formát rovnice je možné využít pouze při volání vestavěných funkcí.

Jazyk obsahuje i vestavěné funkce pro řízení toku při vyhodnocování rovnice – *iif* a *switch*. Podmínky je možné zadávat prostřednictvím logických výrazů.

4.3.2 Operátory

Jazyk kromě aritmetických, logických, bitových a porovnávacích operátorů, známých z jazyka C obsahuje, některé speciální operátory pro veličiny typu signál, definované vlastní logickou tabulkou, například *energetické and* či *energetické or*.

4.3.3 Proměnné

Zásadní odlišností dopočtového jazyka od ostatních procedurálních jazyků jsou proměnné. Proměnné vždy reprezentují konkrétní položky z databáze. Systém *RIS* používá real-time databázový engine postavený na *dbi* tabulkách. Proměnné jsou tedy obecně n-rozměrné vektory, které představují řádek dané tabulky. Jak již bylo zmíněno v kapitole 3.2.2, v dopočtech jsou využívány energetické veličiny a jejich vlastnosti – hodnota, kvalita, alarm a čas.

K těmto čtyřem rozměrům přibývá ještě pátý a to název, kterým je veličina v databázi označena. Název má tvar složený z více částí oddělených dvojtečkou. Každá část hierarchicky specifikuje danou veličinu. Například u veličiny `cSOK4:W1:U`, `cSOK4` značí českou rozvodnu se jménem SOK (Sokolnice) pracující s 400 kW, `W1` je název sběrnice a `U` je indikace napětí. Veličiny jsou z programu v podobě proměnných přístupné užitím jejího názvu.

4.3.4 Výpočet složek proměnných

Přímý přístup k jednotlivým prvkům proměnných není možný a ve většině případů by nedával velký smysl. Pokud je třeba zjistit něco o veličině například z příznaků alarmu, lze k tomu využít vestavěných funkcí. Jinak se vždy s proměnnými počítá jako s celky. Každá část se počítá podle zvláštních pravidel shrnutých níže.

Pravidla pro počítání alarmu:

1. Do výsledku se přenáší pouze příznak aktivity a úroveň závažnosti. Ostatní příznaky jsou ve výsledku ponechány beze změny.
2. Výsledek rovnice získá nejvyšší z úrovní závažností vstupních parametrů rovnice.
3. Pokud má alespoň jeden ze vstupních parametrů rovnice příznak aktivity nastaven, bude mít příznak aktivity nastaven i výsledek.

Pravidla pro počítání kvality:

1. Pokud není programově nastavena implicitní kvalita, za implicitní se považuje kvalita dopočtená. Jinak se implicitní kvalitou stává kvalita programově nastavená.

2. Kvalita chybná není ve výpočtu kvalit uvažována.

3. Má-li jeden z operandů kvalitu neaktuální a současně ne chybnou, výsledek má kvalitu neplatnou.

4. Pokud mají oba operandy kvalitu aktuální, výsledek má kvalitu implicitní.

V případě aktuální kvality všech operandů, platí popsaná pravidla stejně pro všechny funkce a operátory. Pokud má ale některý operand kvalitu neaktuální, chování jistých vestavěných funkcí se od popsaných pravidel může lišit.

Počítání s časy je poměrně jednoduché, výsledek dostane nejvyšší čas z časů, které mají vstupní parametry rovnice.

Pravidla pro počítání hodnoty:

1. Pokud mají všechny vstupní parametry kvalitu jinou než chybnou, výsledná hodnota je spočtena podle předpisu popsaného danou rovnicí.

2. Pokud má vstupní parametr kvalitu chybnou, je z výpočtového předpisu rovnice vypuštěn. (Většinou u operátorů, například rovnice: $a = b+c$, kde c má kvalitu chybnou. Výsledek je b.)

3. Pokud nelze vstupní parametr s chybnou kvalitou vypustit, je výsledná hodnota nedefinována a výsledek dostane chybnou kvalitu. (Většinou u vestavěných funkcí, například rovnice: $a = pow(b, c)$, kde b má kvalitu chybnou. Výsledek je nedefinovaná hodnota s kvalitou chybnou.)

Tato pravidla popisují implicitní chování při výpočtu. Chování je však možné dále různě modifikovat vestavěnými funkcemi. Existuje například funkce *calc_mask* umožňující maskovat zápis jednotlivých prvků do výsledku.

4.3.5 Zpřístupnění databázových tabulek

Před použitím proměnných je nutné nejdříve definovat zdroj, který určuje databázovou tabulku a její sloupce. Úmyslně byl využit název *zdroj* (v programu klíčové slovo `source`). Význam tohoto pojmu vysvětluje úvodní kapitola 3.1. Zde použití názvu `source` značí, že hodnoty veličin jsou do dopočtových rovnic dodávány z databázových tabulek (tabulky jsou tedy zdrojem pro dopočty). Zdroj v programu vymezí umístění a rozměr proměnných. Definice rozměru má význam pouze pro případ, že by uživatel nechtěl pracovat s kvalitou, alarmem nebo časem. Je ovšem povinná i v případě, že mají být využity všechny rozměry. V definici není možné vynechat hodnotu, protože počítat dopočet bez hodnot by nemělo žádný smysl. Následující řádek zdrojového kódu dopočtů definuje zdroj se jménem `A` zpřístupňující sloupce `value` (hodnota), `quality` (kvalita) a `timestamp` (čas) z databázové tabulky `/rdb/analog` (*dbi* tabulka se jménem `analog` patřící lokálnímu *cmi* serveru `rdb`). Sloupec určující názvy veličin v této tabulce se nazývá `name`.

```
source A = '/rdb/analog', name, 'value quality timestamp';
```

U každé proměnné je třeba uvést zdroj, ze kterého pochází (jako prefix jména proměnné oddělený dvojtečkou). Například pro zdroj a veličinu uvedené jako příklady výše je to řetězec `A:cSOK4:W1:U`. Pokud zdroj není u proměnné uveden, uvažuje se implicitní. Po definování se zdroj automaticky stává implicitním. Implicitní zdroj lze též programově přepínat.

Proměnné umožňují i vkládání nových prvků do databáze. Za tímto účelem jazyk poskytuje speciální prefixový operátor @. S jeho využitím lze interpretu zdělit, že pokud veličina s daným jménem v tabulce dané zdrojem neexistuje, má být do tabulky automaticky vložena. Případně je možné interpretu při zavedení dopočtového souboru nastavit příznak, že toto chování má být implicitní. Do databáze jsou pak vloženy všechny nenalezené veličiny.

4.4 Permanentní zdroje

Jak již bylo zmíněno v kapitole 3.2.2, pokud veličina přestane být dodávána do systému (přesněji do *Rdb*), získává kvalitu neaktuální, konkrétně neplatnou. Stejně je tomu i u dopočtů. Ve chvíli, kdy jsou dopočty odinstalovány, všechny platné veličiny získané z dopočtových rovnic jsou prostřednictvím kvality zneplatněny. Tomuto chování lze předejít s využitím speciální části ve formátu v definici zdroje. Všem veličinám z tabulky zpřístupněné daným zdrojem tak zůstane kvalita, kterou získaly v průběhu dopočtu i po odinstalování dopočtového souboru. Takový zdroj veličin je označován jako *permanentní*.

4.5 Spouštění externích programů

Jazyk umožňuje prostřednictvím vestavěné funkce *cmd* spouštět externí programy. Způsob jejich provedení je ovšem dosti neobvyklý. Pro skutečné provedení externího programu je třeba speciální démon *Dcccmd*. Ten poskytuje frontu typu FIFO, do které je možné vkládat příkazy pro spuštění. Funkce *cmd* pouze vloží daný program do fronty čekajících příkazů. Démon frontu prochází a příkazy postupně provádí. Další příkaz provede vždy až po ukončení předchozího. Řetězený způsob prováděný je nutný vzhledem k specifickému použití příkazu *cmd*. Ten se v jazyce využívá pouze pro spuštění klidového, případně fixního způsobu zpracování veličiny v dopočtech.

Klidový mód souvisí s alarmem veličiny. Pokud je veličina v klidovém módu, příznaky alarmů neaktivují uživatelské rozhraní. Fixní mód souvisí s hodnotou veličiny. Pokud je veličina ve fixním módu, *Rdb* ignoruje změny hodnot přicházející z telemetrických měření. Toho se využívá například v případě testování měřiče. Aby nebylo nutné měřič odpojovat ze sítě, veličina se nastaví do fixního módu a odhadnutá hodnota veličiny se zadá ručně. Následné změny, přicházející od testovaného měřiče, pak nejsou ukládány.

Oba tyto speciální módy se spouštějí pomocí programu *rdbalroper*, který slouží pro zasílaní různých druhů zpráv *Rdb*. Funkce *cmd* proto spouští vždy pouze tento program. Očekávaný paralelní způsob provádění spuštěných programů by v tomto případě neměl žádný smysl, protože všechny programy by stejně musely čekat, než je obslouží *Rdb*. Výsledkem by bylo pouze zahlcení systému velkým množstvím čekajících procesů.

Příklad ukazuje použití funkce *cmd* pro aktivaci nebo deaktivaci klidového módu nad veličinami začínajícími `gAGC:SL:s_`. O typu příkazu rozhodne hodnota prvního parametru funkce *cmd*. Hodnota tohoto parametru je promítnuta do řetězce pomocí funkce jazyka C *sprintf*. Pokud bude mít veličina hodnotu 1, výraz předaný jako první argument bude mít též hodnotu 1. V opačném případě bude hodnota výrazu rovna nule. Výsledek výrazu pak v řetězci nahradí formátovací znak `%d`. S využitím argumentu `-k` je tak programu *rdbalroper* sděleno, zda má klidový režim veličin aktivovat či deaktivovat.

```
cmd(K:RIS:BEZ2:s_KIND == 1, "&rdbalroper -k%d -F gAGC:SL:s_*");
```

Poznámka: Úvodní znak `&` v řetězci udávajícím příkaz pro provedení je nutný pro správnou interpretaci funkce *cmd*. Z důvodu zpětné kompatibility je v jazyce možné zadávat proměnné uzavřené v uvozovkách. S využitím znaku `&` lze interpretu sdělit, že se jedná o skutečný řetězec, nikoli o proměnnou. Toto zvláštní chování nebylo z interpretu doposud odstraněno i přesto, že proměnné, tvářící se jako řetězce, nejsou již nevyužívány.

4.6 Chyby překladu a interpretace

Kromě běžných syntaktických a sémantických chyb se v jazyce mohou vyskytnout i chyby datové, tedy chyby, které nezávisí na obsahu zdrojovém souboru, ale na stavu databáze ve chvíli překladu. Konkrétně jde o chyby v definici zdroje – nedostupná tabulka, neexistující sloupce a chybu při použití proměnné – položka s daným jménem v tabulce zpřístupněné daným zdrojem neexistuje. Ve chvíli překladu jsou tyto chyby pouze informativní. Je ovšem nutné je kontrolovat i v interpretu ve chvíli zavedení dopočtu a případně dopočtový binární soubor odmítnout.

4.7 Požadavky na rychlost interpretace

Jak je zmíněno již v kapitole 3.2.1, uživatelské dopočty jsou velmi objemné. V případě dopočtů periodických to nepředstavuje pro interpret velký problém. Nejnižší využívaná perioda je totiž jedna sekunda (nižší ani nemůže být nastavena). Vzhledem k počtu instrukcí, které dnešní počítače během jedné sekundy zvládají, nejsou rychlostní požadavky na interpret nijak zásadní. Problém může nastav v případě dopočtů změnových. Frekvence změn samozřejmě není nijak limitována, takže během jedné sekundy mohou nastat i tisíce změn, které aktivují výpočet rovnic. Navíc je nutné hlídat cyklické závislosti.

4.8 Nedostatky jazyka

Přestože je jazyk využíván již dosti dlouhou dobu, obsahuje řadu nedostatků, které značně znepříjemňují práci uživatelům. Především je to absence uživatelských funkcí, která společně s nemožností přiřazení výstupní hodnoty rovnice více proměnným, nutí uživatele k častému textovému kopírování, což vede ke zbytečným chybám. Další nepříjemností je nemožnost definování pomocných proměnných, či konstant. Dopočtové rovnice jsou proto často nepřehledné a velmi dlouhé.

4.9 Stav jazyka

Dopočtový jazyk je nedílnou součástí systému *RIS* již téměř 15 let. Během této doby se požadavky uživatelů na dopočty měnily. Do jazyka byly postupně přidávány nové funkce, chování funkcí původních bylo různě upravováno. Typickým příkladem je výše zmiňovaná funkce *cmd*. Původně obecná funkce pro spouštění externích programů neměla v dopočtech jiné využití než pro volání služeb *Rdb*. V tomto případě bylo ale chování funkce *cmd* nevhodné, a proto byl zaveden démon *Dcccmd* a tím spouštění externích programů serializováno. Podobných úprav či rozšíření obsahuje jazyk celou řadu. Tyto změny s sebou často přinesly snížení rychlosti interpretace. Tento problém se projevil jako zásadní až s rostoucím objemem rovnic ve změnových dopočtových souborech.

Překladač je implementovaný v jazyce C bez využití nástrojů pro generování zdrojového kódu podle gramatiky jazyka. Vzhledem k průběžným změnám v gramatice jazyka, které byly většinou implementovány v časové tísni, se dostal zdrojový kód překladače do nadále obtížně udržovatelného stavu. To vedlo společně s místy problematickou rychlostí interpretace k rozhodnutí překladač i interpret rekonstruovat.

Kapitola 5

Rozšíření jazyka

Kapitola popisuje rozšíření zahrnutá do jazyka v rámci jeho rekonstrukce.

5.1 Hlavní cíl úprav

Hlavním cílem úprav jazyka bylo především zajištění dostačující rychlosti interpretace. Avšak díky kompletní rekonstrukci bylo vhodné tyto úpravy spojit zároveň s odstraněním nedostatků, které práci s dopočtovými rovnicemi znepříjemňují. Vzhledem k tomu, že se jedná o jazyk komerční, určující byly požadavky a přání uživatelů. Začlenění požadovaných úprav do jazyka ovšem stály v cestě jistá omezení.

5.2 Požadavky uživatelů

Hlavní body se vesměs shodují s nedostatky shrnutými v kapitole 4.8. Uživatelé požadovali zavedení možnosti používat pomocné proměnné. Požadovali proměnné jak perzistentní (zachovají si hodnotu, která jim zůstala ve chvíli odinstalace konkrétního dopočtového souboru), tak běžné pomocné proměnné existující pouze v průběhu interpretace. Dalším požadavkem bylo přidat do jazyka definice uživatelských funkcí a zároveň nějakým způsobem umožnit užívání funkcí již jednou implementovaných v jiném dopočtovém souboru. Tím by se již navždy zamezilo dosavadní nutnosti textově kopírovat úseky zdrojových textů dopočtů. Velmi se rozšířilo používání již zmiňované funkce pro maskování zápisu jednotlivých prvků proměnných. To vedlo na požadavek přidat nový přiřazovací operátor, který provede přiřazení všech prvků veličiny bez ohledu na předchozí maskování.

Přesto, že byli uživatelé důrazně vyzváni, aby dali všechny své požadavky dohromady ještě před začátkem implementace nového překladače a interpretu, poslední požadavky na rozšíření přišly chvíli před dokončením této práce. Jeden požadavek je spojen se schopností některých vestavěných funkcí zpracovávat proměnný počet parametrů. Místy jsou využívány desítky i stovky parametrů a tak je jejich vypisování velmi náročné. Uživatelé tedy vyžadují možnost zápis zkrátit do tvaru textového filtru, na jehož syntaxi jsou zvyklí z uživatelských rozhraní systému *RIS*. Syntaxe filtru bude vysvětlena dále.

Prozatím poslední požadavek je ale mnohem závažnějšího rázu. Jedná se o začlenění takzvaných *blokovacích podmínek* do dopočtového jazyka. Blokovací podmínky jsou v současnosti stále spíše ve fázi návrhu, proto je zde nebudeme dále zmiňovat. Bude jim věnován krátký úsek v kapitole zabývající se budoucím vývojem jazyka.

5.3 Omezení spojené s kompatibilitou

Vzhledem k velkému počtu již aktivních dopočtů bylo nutnou podmínkou dodržet 100% zpětnou kompatibilitu. Případně společně s novou verzí překladače a interpretu poskytnou plně automatizovaný nástroj pro konverzi případných nadále již nepodporovaných konstrukcí. Úpravy jazyka jsou ale pouze rozšiřujícího a optimalizačního charakteru, takže tento problém nebylo nutné řešit.

Při studiu dopočtů a jejich využití se ale zásadním problém ukázala kompatibilita při spolupráci dopočtů s programem zpracovávajícím statistiky, zvaným *Sta*. Program *Sta* je velmi komplexní a není předmětem studia této práce. Proto zde zmíníme pouze pár stručných faktů o statistikách, které poslouží k nastínění vzniklého problému.

5.3.1 Statistika a dopočty

Statistika přestavuje třetí rozměr dvojrozměrných tabulek energetických veličin spravovaných *Rdb*. Tímto třetím rozměrem je čas. Do statistik jsou ukládány stavy veličin v daných časových okamžicích. Veličiny uložené ve statistikách je možné zpětně upravovat. To má význam například v případě zpětně zjištěných poruch telemetrických měřičů či jiných závad. Kromě tohoto hlavního úkolu v systému obstarává program *Sta* spoustu dalších neméně důležitých věcí. Za účelem popisu vzniklého problémů je podstatná pouze jedna její činnost, a to počítání očekávaných mezí ze zaznamenaných hodnot.

Dopočty jsou ve statistice využívány ve dvou zcela odlišných případech. Prvním je dopočet zpětně upravených hodnot. Společně s tabulkami veličin ukládá *Sta* i změnové dopočty, které nad nimi ve chvíli uložení probíhají. Pokud je veličina zpětně upravena, je spuštěn i dopočet. V tomto případě se jedná o běžný změnový dopočet, jak byl doposud představen. Zpětné dopočty jsou z důvodu vysoké náročnosti ukládány na velmi krátké časové úseky v řádu hodin. Poté již úprava dopočet nespustí. Periodické dopočty statistika v tomto případě ignoruje.

Druhým případem užití dopočtů je právě výpočet očekávaných mezí. *Sta* ze zaznamenaných hodnot počítá meze, do kterých by se následující hodnota veličiny měla vejít, pokud nedojde k nějaké mimořádné události. Pro tento výpočet je mimo jiné nutné znát způsob, jakým byla hodnota veličiny získána. Podle toho se poté provádí příslušný výpočet. U veličin dopočtených je třeba znát operace vedoucí k výpočtu hodnoty. Podle příslušných operací provádí *Sta* operace jiné. Často se jedná o operace inverzní, ale ani to není pravidlo. Způsob přístupu statistik k dopočtovému binárnímu souboru je v tomto případě zcela odlišný od běžných dopočtů – *Sta* nepoužívá pro jeho načtení a následnou práci s ním interpret.

Tento zásadní konceptuální nedostatek systému dopočtů vznikl v počátcích tvorby programu *Sta* a doposud nebyl odstraněn. Podle popisu autorů programu *Sta* znemožňuje rozšíření instrukční sady interpretu a velmi omezuje modifikaci formátu binárního souboru, který je generován překladačem. Aby bylo možné úpravy instrukční sady do jazyka zavést bez újmy na funkčnosti dopočtů, bylo by nejdříve nutné změnit koncepci práce s dopočty ve statistikách. To ale v současné situaci především z časových důvodů není možné. Změny, které bylo možné provést v případě binárního souboru jsou vysvětleny v kapitole popisující jeho formát 6.5.1.

5.4 Návrh řešení požadavků s přihlédnutím k omezením

Výše zmiňovaný problém spolupráce dopočtů a statistik limitoval možnosti rozšíření a úprav jazyka. Po zvážení situace a konzultacích s autory programu *Sta* i s uživateli jazyka se jako nejlepší možné řešení jevilo zavedení textového preprocesoru umožňujícího definice maker.

Textová makra by přinesla alespoň částečnou náhradu za uživatelské funkce, jejichž implementaci program *Sta* znemožnil. Makra by zároveň umožnila definice symbolických konstant.

Požadavek na používání funkcí (nyní již pouze maker) definovaných v jiných dopočtových souborech by bylo možné řešit opět preprocesorem. Stačilo by implementovat textové vkládání externích zdrojových souborů, jak je běžné v jiných programovacích jazycích.

Pro splnění požadavku na definice pomocných proměnných se nabízelo jednoduché řešení. Vytvoření nové databázové tabulky jako součást *Rdb*, která je následně v programu zpřístupněna jako běžný zdroj. Operátorem @ je potom možné do tabulky vkládat nové proměnné.

Nový přiřazovací operátor nepředstavoval závažnější problém. Jde o úpravu gramatiky jazyka, která není pro program *Sta* nijak zásadní. Vlastní funkčnost operátoru bylo možné vyřešit s využitím funkce *calc_mask*, která již byla součástí instrukční sady původního interpretu.

Naproti tomu používání textových filtrů pro zadávání výčtu proměnných představuje pro spolupráci se statistikou problém značný. Pro pochopení daného problému je nejdříve nutné vysvětlit, co to vlastně textový filtr je.

Textový filtr je podporován rozhraním *dbi*. Podmínkou je, že sloupec, nad kterým je definován index určený pro filtrování, musí být typu řetězec. Filtr pak umožňuje prostřednictvím textového řetězce vyjmenovat jména, která mají být vybrána. V řetězci je možné používat různé speciality jako například zástupný znak *, nerozlišování malých a velkých písmen, selekci všechny jmen kromě zadaných a podobně. Po aplikování filtru pomocí kurzoru se tabulka tváří, jako kdyby obsahovala pouze položky, které filtru odpovídají.

Níže uvedený příklad řetězce pro textové filtrování vybere z tabulky všechny veličiny, jejichž název začíná `gAGC`. Do výběru přitom nezahrne veličiny začínající `gAGC:SL` nebo `gAGC:SAL`, a to bez rozlišení velkých a malých písmen (mínus před negovanými jmény).

$$\text{"gAGC}* \;-\; \text{!gAGC:SL}* \;\; \text{!gAGC:SAL}*\text{"}$$

Řešení tohoto problému tedy vyžaduje nahrazení textového řetězce odpovídajícími prvky dané tabulky. Prvky odpovídající filtru lze však zjistit až podle obsahu dané tabulky. Ten však může být odlišný v době překladu zdrojového souboru a v době interpretace přeloženého souboru binárního. Je zřejmé, že v tomto případě se jedná o závislost datovou, která by správně neměla být řešena při překladu, ale až při zavedení dopočtu. To však omezení, daná programem *Sta*, nedovolují. Proto bylo prozatím zvoleno řešení značně omezující, a to vyhodnocování filtru ve chvíli překladu.

Kapitola 6

Návrh a Implementace

Kapitola se zaměřuje na návrh a následné řešení preprocesoru, překladače a interpretu. Popisuje implementační detaily jednotlivých částí společně s problémy, které bylo nutné řešit. V této kapitole se bude práce věnovat čistě vlastní tvorbě autora.

6.1 *Dbi* tabulky jako základní datové struktury

Již bylo vysvětleno, že pro uložení perzistentních dat slouží v systému *RIS* univerzální tabulkový interface – *dbi*. Protože se ale tabulky nacházejí kompletně v operační paměti a poskytují celou řadu operací, jsou využívány i jako základní datové struktury v programech, kde není případná perzistence vůbec potřebná. Využívají se například jako implementace asociativních polí, ale i jako běžná pole či zásobníky struktur. Za tímto účelem existuje speciální druh indexu *native*. Prvky do tabulky s indexem *native* jsou zařazovány v tom pořadí, v jakém byly vloženy. Pokud není žádný index definován, je index *native* implicitní. V případě použití tabulek jako běžných datových struktur se samozřejmě využívá přímého přístupu k jednotlivým prvkům (řádkům tabulky).

Při implementaci preprocesoru, překladače a interpretu bylo využito tabulek pro řešení většiny problémů.

6.2 Rozdělení projektu na celky

Kvůli již několikrát zmiňované závažnosti úloh řešených dopočty, bylo od počátků jasné, že nedílnou součástí práce bude muset být kvalitní testování. Případné chyby by mohly mít velice nepříjemné následky. Úvodním krokem při návrhu bylo tedy rozdělení na samostatně testovatelné celky. Vhodné se jevilo dělení následující:

1. preprocesor
2. lexikální a syntaktický analyzátor
3. generátor binárního kódu
4. interpret

6.3 Preprocesor

Především značná syntaktická podobnost dopočtového jazyka s jazykem C vedla k rozhodnutí zachovat syntaxi jazyka C i při návrhu preprocesoru. Jako referenční řešení sloužil preprocesor překladače *gcc* (viz. [7]). Požadavky na preprocesor byly textová makra s parametry a vkládání externích souborů do zdrojového textu. Vzhledem k zavedení maker a externích souborů bylo pravděpodobné, že by se brzy objevil také požadavek na řešení problémů jako nežádoucí redefinování již definovaných maker. To vedlo k rozhodnutí zahrnout do preprocesoru v omezené míře i podmíněný překlad. Odstranění komentářů ze zdrojového kódu bylo opět podle vzoru jazyka C přeneseno z lexikálního analyzátoru do preprocesoru.

Úkolem preprocesoru je tedy zpracovat direktivy *include*, *define*, *undef*, *ifdef*, *elif*, *else* a *endif*, zbavit zdrojový kód komentářů a nějakým způsobem sdělit lexikálnímu analyzátoru výsledek úprav, které změnily pořadí řádků, aby bylo možné se při výpisu případných chyb odkazovat na správné místo ve zdrojovém souboru.

6.3.1 Flex a startovací stavy

Pro implementaci preprocesoru bylo využito nástroje Flex, který významným způsobem ulehčil práci. Ten umožňuje vygenerovat konečný automat na základě popisu pravidel regulárními výrazy. Nejdříve byla zvažována i možnost využít kombinace Flex a Bison, která by usnadnila řešení některých kontextových závislostí jako je formát definice maker. Nakonec bylo ale pro řešení kontextových závislostí využito *startovacích stavů* (viz. [5]), které nabízí Flex, a tak použití Bisonu nebylo nutné. Prostřednictvím *startovacích stavů* nebo také *startovacích podmínek* je možné specifikovat, které regulární výrazy je možné v danou chvíli v textu vyhledávat. Pravidla v podobě regulárních výrazů, která jsou označena daným startovacím stavem, budou v textu vyhledávána pouze pokud se konečný automat v tomto startovacím stavu nachází.

Použití startovacích stavů s sebou přineslo i menší problém. Pravidla využívající startovacích stavů musejí v sobě zahrnovat i případné komentáře, protože regulární výrazy pro blokové a řádkové komentáře jsou dostupné pouze ve stavu inicializačním.

6.3.2 Vkládání externích souborů

Řešení tohoto problému využívá možnosti Flexu pracovat s více vstupními soubory najednou. Pomoci funkcí *yy_create_buffer*, *yypush_buffer_state* a *yypop_buffer_state* lze ukládat vstupní soubory do interního zásobníku a následně je z něj načítat. To je přesně chování, které je potřebné pro implementaci vkládání externích souborů. Ve chvíli rozpoznání direktivy *include* je aktuálně zpracovávaný soubor uložen na zásobník a přejde se na zpracovávání nově otevřeného. Jakmile je přečten konec souboru, je z vrcholu zásobníku vyzvednut naposledy uložený soubor a zpracovávání pokračuje.

6.3.3 Textová makra

Zřejmě nejobtížnějším úkolem řešeným v rámci preprocesoru byla makra. Při implementaci byla využita celá řada startovacích stavů, pomocí kterých byly vyřešeny kontextově závislé konstrukce a to jak v definici, tak následně při vyhledávání odpovídajících identifikátorů ve zdrojovém textu.

Implementace využívá celkem čtyři tabulky. První, sloužící k uložení definovaných maker, s indexem nad názvem makra pro rychlému asociativními vyhledávání. Druhá, vnořená v tabulce maker, určená pro uložení případných formálních parametrů, třetí pro ukládání parametrů skutečných při hledání maker ve zdrojovém souboru. Obě tabulky určené pro ukládání parametrů mají index native, proto jsou vždy odpovídající si dvojice parametrů na stejným pozicích. Poslední tabulka slouží k průběžnému ukládání právě generovaných maker z důvodu detekce maker odkazujících se sama na sebe. Tento problém bude vysvětlen v následující podkapitole. Tabulka má definovaný index nad jménem makra opět z důvodu asociativního vyhledání.

Ve chvíli expanze maker s parametry, je třeba v těle makra nahradit formální parametry skutečnými. Algoritmus pro náhradu parametrů musí projít obsah makra a kdykoli narazí na některý z formálních parametrů, provede jeho náhradu za parametr skutečný. Následně pokračuje v procházení až za nahrazeným skutečným parametrem. Substituce je tak provedena pouze v jednom průchodu. S použitím jednoduchého řešení, kdy by se nahradily nejdříve všechny výskyty jednoho formálního parametru a následně se začaly nahrazovat výskyty parametru dalšího, by mohlo dojít k chybě. Skutečný parametr, kterým by již byl první formální parametr nahrazen, by totiž mohl obsahovat formální parametr druhý a to by vedlo k nežádoucí náhradě.

Samotná expanze maker (náhrada identifikátoru reprezentujícího makro za jeho obsah ve zdrojovém textu) byla implementována trikem převzatým z [10]. Využívá funkce *unput* poskytnuté Flexem, která dokáže vrátit načtený znak do bufferu, pomocí kterého konečný automat přistupuje k aktuálně zpracovávanému souboru. Tuto funkci je možné volat i několikrát v řadě za sebou, a to i pro znaky, které nebyly přečtené ze vstupního bufferu. Lze tak do bufferu vložit například vlastní řádek, který je následně zpracován konečným automatem, jako kdyby byl součástí vstupního souboru. Jak se uvádí ve výše citované literatuře, Flex je schopen vložit zpět do bufferu minimálně tolik znaků, kolik obsahuje nejdelší načtený token. Díky tomu, že je celý obsah makra načten jako jeden token, je možné funkci *unput* využít při jeho expanzi. Ve chvíli, kdy je ve zdrojovém kódu rozpoznáno makro, se nejdříve provede náhrada formálních parametrů skutečnými a potom je jeho obsah vložen do bufferu. To jednoduše zajistí správné rozpoznání dalších maker uvnitř obsahu makra prvního. Toto řešení ovšem přináší problém v případě maker odkazujících se sama na sebe.

6.3.4 Makra odkazující se sama na sebe

Makra odkazující se sama na sebe, jsou makra, která mají ve svém těle použito svoje jméno. Například:

```
#define A       A + 3
```

Dalším případem jsou makra nepřímo odkazující se na sebe. Například:

```
#define A       DEFINE B
#define B       DEFINE A
```

V obou případech by běžná expanze skončila nekonečnou rekurzí. Jak se píše v manuálu překladače *gcc* v sekci popisující makra odkazující se sama na sebe (viz. [6]), má být makro rozvinuto, pokud je součástí těla jiného makra, nikoli však jestliže je součástí svého vlastního těla a to jak přímo, tak nepřímo. V tomto případě je makro vypsáno na výstup nezměněné.

Vzhledem ke zvolenému způsobu expanze maker nelze nijak přímo rozlišit, zda je ze vstupního bufferu čteno již expandované makro nebo obsah zdrojového souboru. Za tímto účelem byla vyhrazena tabulka aktuálně generovaných maker. Tabulka obsahuje dvojice

ve tvaru jméno – počítadlo. Ve chvíli expanze je makro do této tabulky vloženo a počítadlo nastaveno na délku obsahu zvětšenou o jedna. Zároveň jsou o délku obsahu inkrementována počítadla všem záznamům, které tato tabulka již obsahuje. Při přečtení jakéhokoli tokenu je všem záznamům uloženým v tabulce generovaných maker dekrementováno počítadlo o délku tohoto tokenu. Jestli má být makro z tabulky odstraněno je dáno testem počítadla na nulovou nebo zápornou hodnotu po dekrementaci. Pak už pouze stačí při každé expanzi zkontrolovat, zda dané makro není v tabulce aktuálně generovaných maker a případně expanzi nepovolit.

Po vzoru jazyka C, umožňují makra také konkatenaci parametrů s textovým řetězcem.

6.3.5 Podmíněný překlad

S využitím podmíněného překladu je možné preprocesoru sdělit, aby specifikované části zdrojového souboru nepřenášel do souboru předzpracovaného. Tato schopnost preprocesoru nebyla uživateli požadována, a proto byla oproti jazyku C implementována pouze zjednodušeně. Preprocesor umí zpracovat všechny podmíněné direktivy s výjimkou *if* a *defined*. Parametrem implementovaných direktiv může být pouze jeden identifikátor reprezentující jméno makra. Žádné složené výrazy prozatím nejsou podporovány.

Pro ukládání podmínek slouží *dbi* tabulka, která je v tomto případě využita jako zásobník. Kromě pravdivostní hodnoty podmínky jsou totiž ukládány i další informace sloužící pro případné výpisy chyb.

Ve chvíli zpracování podmíněné direktivy je vyhodnocena podmínka – ověření, zda makro se zadaným názvem bylo definováno. V případě, že na vrcholu zásobníku je již negativně vyhodnocená podmínka, není třeba existenci makra ani ověřovat a podmínka je přímo vyhodnocena jako negativní. Tato podmínka je následně společně s dalšími informacemi nutnými pro případná chybová hlášení uložena na zásobník. Preprocesor při zpracovávání zdrojového souboru kontroluje tento zásobník podmínek. Pokud je na jeho vrcholu podmínka negativní, ignoruje obsah zdrojového souboru, kromě direktiv podmíněných. Podmíněné direktivy je třeba zpracovávat a ukládat na zásobník i v případě platnosti negativní podmínky. Toto chování je nutné z důvodu detekce konce platnosti uložených podmínek.

Podle preprocesoru jazyka C není význam podmíněných direktiv přenášen mezi různými soubory. Je-li tedy přečten konec souboru a zásobník podmínek není prázdný, jsou všechny neuzavřené podmíněné direktivy hlášeny jako chyby. Zásobník je následně vyprázdněn.

6.3.6 Komunikace preprocesoru a lexikálního analyzátoru

Posledním úkolem preprocesoru je odstranění řádkových komentářů. V tomto případě se nejedná o implementačně náročný problém. Tato akce, stejně jako další akce prováděné preprocesorem, může změnit počet řádků zdrojového souboru. Výrazným způsobem mění čísla řádků textové vkládání externích souborů, které navíc mění i zdrojový soubor. Tento fakt musí preprocesor sdělit lexikálnímu analyzátoru z důvodu případných chybových hlášení. Chybová hlášení je třeba odkazovat na správné řádky ve zdrojovém souboru. U souborů vložených musí být chybové odkazy směrovány do správného zdrojového souboru, nejlépe i s cestou, která vedla k vložení tohoto souboru. V případě zanořeného vkládání, které není žádnou výjimkou, je výpis cesty téměř nezbytný. Různé implementace preprocesorů běžně tento problém řeší pomocí značek vložených do předzpracovaného souboru. Tyto značky sdělují lexikálnímu analyzátoru, jak si má upravit interní počítadlo čísla řádku či název aktuálního zpracovávaného souboru. Referenční preprocesor překladače *gcc* řeší tento problém

poměrně kvalitně. V případě většího počtu prázdných řádků, které by vznikly v předzpracovaném souboru nebo už jsou v souboru zdrojovém, tyto řádky do výstupního souboru nezařadí. Následná značka sděluje na jakém řádku kterého zdrojového souboru se lexikální analyzátor aktuálně nachází. Výstupem tohoto řešení je pak přehlednější předzpracovaný soubor. V implementaci preprocesoru bylo zvoleno řešení jednodušší. Preprocesor v případě vynechání řádku zdrojového souboru vloží na jeho místo řádek prázdný, a tak ponechá počet řádků nezměněný. Jediné značky nutné pro komunikace s lexikálním analyzátorem jsou tedy označení začátku a konce vloženého zdrojového souboru.

6.4 Lexikální a syntaktický analyzátor

Přestože se jedná o základní částí překladače, stále jsou spíše v rozpracovaném stavu. Hlavní snaha úprav byla od počátku směřována k návrhu a implementaci kvalitního nového interpretu. Proto byl zvolen postup lexikální a syntaktický analyzátor stejně jako preprocesor implementovat co možná nejrychleji, a poskytnout tak základnu pro tvorbu binárního kódu. Tak by bylo možné začít se věnovat interpretu a s ním souvisejícímu generátoru binárního kódu. K dokončení kvalitního preprocesoru, lexikálního a syntaktického analyzátoru se pak vrátit až po implementaci interpretu. Ve chvíli úprav preprocesoru se však objevily nové požadavky na jazyk, a tak bylo dokončení lexikálního a syntaktického analyzátoru odloženo.

Nedostatky lexikální a syntaktické analýzy jsou především v neschopnosti podávat přesnější chybové hlášení. Obě části jsou prozatím implementovány pouze tak, aby přijímaly jazyk popsaný gramatikou dopočtového jazyka a v případě chybného vstupu bylo možné se z chyby zotavit a pokračovat ve zpracování dále. Jediné chybové hlášení vypisované syntaktickým analyzátorem je pouze nic neříkající „syntax error“ s příslušným názvem souboru a číslem řádku.

6.4.1 Generování zdrojového kódu podle gramatiky

Při tvorbě lexikálního a syntaktického analyzátoru bylo rozhodnuto využít možnosti vygenerování zdrojového kódu pomocí zadané gramatiky. Vedly k tomu především dva důvody. Za prvé, během doby, po kterou je jazyk používán, se již několikrát objevila nutnost upravit jeho gramatiku. Je tak velmi pravděpodobné, že budoucí požadavky uživatelů povedou k úpravám dalším. Začlenění úprav gramatiky do vlastnoručně naprogramovaného překladače je téměř vždy časově náročnou záležitostí. Oproti tomu přidání, či změna některého pravidla gramatiky podle nutných požadavků, a následné přegenerování syntaktického analyzátoru, který ji implementuje, je většinou úprava snadná. Druhým důvodem byla značná rozsáhlost projektu. Časová úspora, kterou poskytují nástroje pro automatické vytvoření lexikálního a syntaktického analyzátoru, je nesporná.

Za tímto účelem bylo využito kombinace utilit Flex a Bison. Bison umožňuje vygenerovat LALR syntaktický analyzátor (viz. [13]) na základě pravidel gramatiky popsané Backus-Naurovou formou. Flex posloužil pro vytvoření konečného automatu zpracovávajícího vstupní soubor na proud tokenů z kterých syntaktický analyzátor vygenerovaný Bisonem tvoří věty podle gramatiky jazyka.

6.4.2 Tvorba gramatiky

Gramatika jazyka nebyla nikde sepsána, a tak bylo nutné ji nejdříve získat z původního syntaktického analyzátoru. To se ovšem ukázalo jako téměř nemožné. Časté úpravy učinily již

tak poměrně složitý zdrojový kód syntaktického analyzátoru velmi obtížně pochopitelným. Mnohem příznivější cestou bylo studium zdrojových kódů dopočtových souborů a případné konzultace s tvůrci původní verze překladače. Touto metodou reverzního inženýrství byla vytvořena sada pravidel v Backus-Naurově formě vyžadovaných Bisonem pro vygenerování syntaktického analyzátoru.

6.4.3 Syntaktický analyzátor jako knihovna

Systém *RIS* využívá kromě popisovaných typů dopočtů ještě dopočty tabulkové. Jedná se o předchůdce dopočtů změnových a periodických, který je ale stále z důvodu zpětné kompatibility na některých místech využíván. Tabulkové dopočty mají stejnou syntaxi, jako dopočty diskutované v této práci. Z tohoto důvodu byl původní syntaktický analyzátor implementován jako knihovna, které je nutné před použitím poskytnou konkrétní callbackové funkce volané v případě příslušné akce. Aby nebylo nutné původní syntaktický i lexikální analyzátor ponechávat součástí systému, bylo rozhodnuto nový syntaktický analyzátor implementovat podobným způsobem. Registrace velkého množství callbackových funkcí jako součást syntaktického analyzátoru je ale značně nepříjemná, a tak je implementace požadovaného chování lehce „ochuzená“. Syntaktický analyzátor je vytvořen jako knihovna, která volá neimplementované funkce. Poskytuje pouze soubor rozhraní s prototypy požadovaných funkcí. Implementaci funkcí dodá konkrétní překladač, který bude chtít knihovnu využívat, až ve fázi linkování s knihovnou. U těchto funkcí rozhraní byla záměrně využita jména, která používají tabulkové dopočty. Díky tomu je možné nový syntaktický analyzátor využít i u tabulkových dopočtů téměř bez jejich modifikace.

Knihovna obsahuje i implementovaný preprocesor, který samozřejmě v tabulkových dopočtech nenajde uplatnění. Součástí knihovny je jednoduché rozhraní pro výpisy chybových hlášení odkazující na čísla řádků, případně do externích vložených souborů s cestou, která vedla k jejich vložení.

6.5 Binární kód

Před popisem generátoru binárního kódu bude nejvhodnější představit binární kód samotný. Ten opět využívá *dbi* tabulek. Tentokrát ovšem na jiný způsob. Celý binární kód je ve skutečnosti jedna složitá tabulka obsahující další tabulky vnořené. V této tabulce je uložen pouze jediný záznam. Data s globální platností pro celý zdrojový kód jsou uložena přímo v tomto záznamu – například změněná implicitní kvalita nebo délka periody. Data popisující jednotlivé rovnice či zdroje jsou pak ukládána do tabulek vnořených. V terminologii *dbi* se takováto tabulka nazývá *header*.

Binární soubor je tedy *dbi* tabulka. Tyto tabulky jsou po načtení do paměti ve zdrojovém kódu přístupné jako struktury jazyka C. Zde pramení příčina vzniku problému při spolupráci dopočtů s programem *Sta*. Zápis binárního souboru na disk, stejně jako jeho načítání do programu, provádí funkce rozhraní *dbi*. Autoři programu *Sta* tedy mohli snadno načíst binární soubor a následně s ním pracovat pouze se znalostí prototypu struktury reprezentující v programu *header*. Z toho vyplývá, které změny bylo možné v binární kódu provést, aby byla zachována kompatibilita s programem *Sta*. *Sta* očekává jistý formát struktury, se kterým pracuje. Tento formát nemůže být narušen, ale může být rozšířen podle potřeby. Statistika toto rozšíření při práci s dopočty jednoduše nevyužije. Stejně tak části původní struktury představující binární kód, které program *Sta* nevyužívá, mohou být odstraněny.

Vzniklá omezení se týkají konkrétně přístupu k rovnicím a zdrojům. V těchto podtabulkách musel být dodržen jistý základ.

6.5.1 Formát binárního kódu

Nyní představíme důležité části z formátu binárního souboru ve tvaru definice tabulky, která je zpracována preprocesorem *drcc*, doplněné o stručný popis. V následujících podkapitolách budeme postupně jednotlivé části rozebírat a vysvětlovat.

název	**datový typ**	**popis**
`version`	`INT`	verze binárního souboru
`table`	`CURSOR`	vnořená tabulka zdrojů
`eqc`	`CURSOR`	vnořená tabulka rovnic
`fun`	`CURSOR`	vnořená tabulka uživatelských funkcí
`temp`	`CURSOR`	vnořená tabulka pomocných proměnných
`tln`	`CURSOR`	vnořená tabulka lokalizovaných tabulek
`buf`	`VARBINARY`	výpočtový buffer
`vext`	`VARBINARY`	seznam interních rozšiřujících funkcí
`flg`	`INT`	příznaky s globální platností
`act_ieqc`	`INT`	aktuálně generovaná rovnice
`dfl_itable`	`INT`	implicitní zdroj
`dfl_prefix`	`VARCHAR`	implicitní prefix jména tabulek
`act_time`	`DATETIME`	aktuální čas výpočtu od začátku hodiny
`prev_time`	`DATETIME`	předchozí čas výpočtu od začátku hodiny
`period`	`INT`	perioda výpočtu v sekundách
`quality`	`UINT`	implicitní kvalita

Tabulka 6.1: Formát tabulky představující binární kód

Poznámka: Jak ukazuje příloha A, definice tabulky může obsahovat kromě prvních dvou sloupců určených pro název a datový typ ještě sloupec třetí. S jeho využitím lze specifikovat datový typ jazyka C, kterým bude reprezentována položka struktury. Tento sloupec je nezbytný například u položek typu `VARBINARY` běžně používaných pro definici pole. V tomto případě udává sloupec typ prvků tohoto pole. Tento sloupec zde z důvodu zvýšení přehlednosti neuvádíme.

6.5.2 Odkazy mezi *dbi* tabulkami s perzistentní platností

Nejrychlejší přístup k jednotlivým prvkům *dbi* tabulky je přes její vnitřní indexy – *iid*. Tyto vnitřní indexy jsou po dobu existence prvku neměnné. Od vytvoření nového prvku tedy není *iid* změněno žádnou operací nad prvkem ani nad tabulkou, kromě operace vymazání. Ve chvíli odstranění prvku z tabulky je jeho *iid* uvolněno a může, ale nemusí, být přiděleno prvku novému. Toto vnitřní indexování zůstává neměnné i v případě uložení a následného načtení tabulky z disku. S využitím *iid* je například možné odkazovat se z jedné tabulky na prvek tabulky jiné. Tento odkaz zůstane platný dokud nebude daný prvek vymazán, a to i po uložení a opětovném načtení obou tabulek z disku. Popsaného chování se v binárním kódu často využívá.

6.5.3 Uložení zdrojů

Na začátek připomeneme fakt, že tabulky sloužící k uložení zdrojů, rovnic, funkcí a dalších částí jsou sice implementovány úplně stejným způsobem jako tabulky databázové, ale jedná se o dvě zcela odlišné věci (dva odlišné případy využití univerzálního rozhraní *dbi*). Dále je vhodné zopakovat, že s využitím zdroje se ve zdrojovém kódu specifikuje databázová tabulka, ze které proměnná, reprezentující konkrétní veličinu, pochází.

Kvůli statistice bylo nutné přebrat koncept používaný dříve. Tabulka zdrojů obsahuje další dvě vnořené tabulky. Do první se ukládají prvky, které se ve zdrojovém kódu vyskytují jako vstupní parametry rovnic a pocházejí z databázové tabulky zpřístupněné tímto zdrojem. Ve druhé tabulce jsou uloženy všechny výstupní parametry pocházející z dané databázové tabulky. Kromě těchto dvou základních vnořených tabulek obsahuje tabulka zdrojů ještě řadu dalších údajů, které daný zdroj blíže specifikují, jako je jméno databázové tabulky nebo formát určující používané sloupce.

6.5.4 Uložení rovnic

Formát uložení rovnic byl také přebrán z původní verze. Chování změnových dopočtů jasně určuje podmínku uchovávat jednotlivé rovnice odděleně. Obecně je nutné u každé rovnice znát vstupní parametry, operace nad těmito parametry prováděné a parametry výstupní. Kvůli speciální formě rovnic, představené v kapitole 4.3.1, může být výstupních parametrů více.

Každá rovnice je popsána dvěma poli implementovanými datovým typem `VARBINARY` a skupinou příznaků. V prvním poli jsou uloženy výstupní parametry jako odkazy do tabulky zdrojů ve tvaru: *iid* záznamu v tabulce zdrojů – *iid* záznamu ve vnořené tabulce výstupních parametrů. V poli druhém je uložen výpočtový předpis specifikující rovnici. Prvkem výpočtového předpisu je struktura obsahující typ prvku a data pro konkrétní typ uložená jako unie jazyka C. Unie může obsahovat tyto položky:

- celočíselnou konstantu
- konstantu v plovoucí řádové čárce
- proměnnou
- operátor nebo vestavěnou funkci

Proměnné představují vstupní parametry rovnice a jedná se opět o odkazy do podtabulky vstupních proměnných v tabulce zdrojů s využitím *iid* (*iid* zdroje – *iid* vstupního parametru).

Prvky výpočtového předpisu jsou v poli organizovány v obrácené polské notaci (viz [15]), dále jen RPN (Reverse Polish Notation). Jedná se postfixovou notaci, která umožňuje zápis výrazů bez nutnosti užití závorek pro určení priority operací. Nejdříve jsou zapsány operandy a následně operátor. Pro ilustraci uvádíme dva příklady výrazů zapsaných v RPN:

běžný zápis	RPN
$a + b * c$	$a\ b\ c * +$
$(a + b) * c$	$a\ b + c *$

Tento způsob zápisu lze využít i pro operace, které vyžadují jiný počet operandů než dva. Stačí správný počet operandů specifikovat. Díky uložení rovnic v RPN mohou operátory a vestavěné funkce sdílet stejnou položku unie pro uložení dat, protože u obou je třeba znát pouze typ prováděné operace a počet parametrů vyžadovaných danou operací.

6.5.5 Pomocné proměnné

Požadavek na užívání pomocných proměnných byl nejdříve vyřešen zavedením nové pomocné databázové tabulky jako součást *Rdb*. V případě běžných pomocných proměnných, které jsou platné pouze v průběhu dopočtu, přineslo ale toto řešení komplikace. Tyto proměnné musí být ve chvíli odinstalace dopočtu z pomocné databázové tabulky explicitně odstraněny. Problém nastává ve speciálním zmiňovaném případě, kdy program *Sta* nevyužívá rozhraní interpretu pro práci s binárním souborem, tedy nevolá ani funkci interpretu pro odinstalování dopočtu. V tomto případě není možné proměnné z pomocné databázové tabulky odstranit. Proto bylo pro tyto proměnné zvoleno jiné řešení – zavedení dalšího zdroje přímo jako součást binárního kódu. Tento zdroj představuje vnořená tabulka *headru* `temp`. Záznam o něm je do tabulky zdrojů vložen již při generaci binárního souboru.

6.5.6 Uživatelské a interní rozšiřující funkce

V tabulce 6.5.1 vidíme, že binární kód obsahuje hned dvě možnosti uložení definice funkcí – uživatelské funkce a interní rozšiřující funkce. Jak je však vysvětleno v kapitole 5.3, zavedení uživatelských funkcí nebylo možné. V současnosti nenajdou tyto dvě části žádné využití, jejich zařazení do binárního kódu a částečně i do kódu překladače a interpretu bylo pouze z důvodu pravděpodobné budoucí implementace.

Uživatelskými jsou nazývány funkce běžně definované uvnitř zdrojového kódu. Interními rozšiřujícími jsou funkce, které je možné přidávat do dopočtů bez zásahu do zdrojového kódu překladače a interpretu. To by v budoucnu mohlo najít využití pro funkce jako je například *split*, které byly implementovány pouze pro specifické využití dopočtů. Tyto funkce by pak bylo možné zpřístupnit pouze na těch aplikacích systému *RIS*, pro které byly navrženy. Hlavním důvodem je ale především to, že specifické funkce v minulosti většinou přidávali do dopočtů specialisté na konkrétní problematiku. Jejich zásahy přímo do kódu interpretu ale nejsou žádoucí z důvodu zanesení možných chyb či snížení výkonnosti.

6.5.7 Prefixování jmen tabulek

Díky zpětným dopočtům ve statistikách byl zaveden systém prefixování tabulek. Ve zdrojovém kódu se názvy databázových tabulek zadávají bez prefixu, který určuje název *cmi* serveru (procesu vlastnící dané tabulky). Tento prefix se nastavuje přepínačem až při překladu a instalaci zdrojového souboru. Například pro zpřístupnění tabulky obsahující všechny veličiny typu *analog* z *Rdb* se místo správného názvu databázové tabulky `/rdb/analog` ve zdrojovém kódu uvede pouze `analog` a prefix `/rdb/` doplní překladový a instalační skript. Je to zjednodušení pro uživatele a také pro statistiku, která nemusí prefix u názvů tabulek odstraňovat.

6.6 Generátor binárního kódu

Úkolem generátoru je vytvořit *header*, naplnit ho daty podle obsahu zdrojového souboru a nakonec ho zapsat na disk jako výstupní binární kód.

Generátor pracuje na základě funkcí volaných syntaktickým analyzátorem při rozboru zdrojového kódu. Jak bylo vysvětleno v kapitole 6.4.3, implementaci těchto funkcí musí dodat konkrétní překladač využívající syntaktický analyzátor. Zde byly tyto funkce implementovány jako obálka pro rozhraní generátoru a případné výpisy chybových hlášení podle rozboru návratové hodnoty generátoru. Funkce, která oznamuje výskyt operátoru, ještě před voláním generátoru převádí token reprezentující operaci na kód operace.

6.6.1 Reentrantní rozhraní generátoru

Vzhledem ke koncepci interpretu (rozhraní využívané ostatními programy) je nutným požadavkem funkce interpretu implementovat jako reentrantní (viz. [14]). To vedlo k rozhodnutí implementovat jako reentrantní i jádro generátoru pro případ, že by našla tato užitečná vlastnost v budoucnu využití. Stav generace je tak uložen v generovaném *headru*. Za tímto účelem je nutné uchovávat *iid* aktuálně generované rovnice (v tabulce 6.5.1 `act_ieqc`) a příznaky sloužící pro správné počítání parametrů funkce při použití filtru.

6.6.2 Generace zdrojů

Ve chvíli příchodu požadavku na přidání nového zdroje, generátor nejdříve zkontroluje, zda se již databázová tabulka s požadovaných formátem v tabulce zdrojů nevyskytuje. Jestliže zdroj zpřístupňující databázovou tabulku se shodným formátem nenalezne, nový zdroj do tabulky zdrojů vloží. V opačném případě je hlášena chyba. Předejde se tak duplicitám zdrojů, které by zbytečně zvětšovaly velikost výsledného binární souboru. Končí-li formát speciálním znakem *!*, poznamená se, že se jedná o zdroj permanentní. Tento znak je z formátu odstraněn a následuje kontrola zdroje. Nejdříve se ověří existence databázové tabulky v systému. Následně se zjišťuje, zda tabulka obsahuje sloupce požadované formátem zdroje. Tato kontrola je ale pouze informativní a i v případě jejího selhání může být překlad úspěšně dokončen. Nemusí se totiž nutně jednat o chybu ve zdrojovém kódu, ale pouze o aktuálně nesplněnou datovou závislost.

6.6.3 Převod rovnic do RPN

Při zpracování jednotlivých rovnic je úkolem generátoru převést rovnice zapsané v notaci infixové do RPN, v níž jsou uloženy v binárním kódu. Tato akce však nepředstavuje žádný problém díky skutečnosti, že generace rovnic probíhá na základě funkcí volaných syntaktických analyzátorem.

Syntaktické analyzátory pracující metodou zdola nahoru totiž zpracovávají výrazy způsobem, který umožňuje převod notace infixové na postfixovou velmi snadno. Analyzátor LALR(1) generovaný bisonem mezi ně patří. Při implementaci pravidel pro Bison určených pro zpracování výrazů se stačí držet zásady, že akce oznamující výskyt operátoru či funkce ve zdrojovém kódu je třeba volat na konci pravidla. Operandy tak budou oznámeny dříve než operace. Příklad implementace pravidla:

```
expr : expr '+' expr      { seen_operator( '+' ); }
     | IDENTIFIER         { seen_identifier( $1 ); }
     ;
```

Syntaxe zápisu pravidel pro Bison zde není podstatná. Důležité je, že vygenerovaný syntaktický analyzátor zpracovávající výrazy podle výše popsaného pravidla, pak pro výraz $a + b$ bude volat následující funkce:

1. *seen_identifier* s parametrem *a*
2. *seen_identifier* s parametrem *b*
3. *seen_operator* s parametrem $'+'$

Vidíme, že posloupnost volaných funkcí odpovídá výrazu v RPN. Stejným způsobem bude probíhat i rozbor výrazů složitějších. Požadavky na generaci tak budou do generátoru přicházet v pořadí, které odpovídá přepisu zpracovávané rovnice do RPN.

6.6.4 Generace rovnic

Před začátkem zpracovávání rovnice je syntaktickým analyzátorem volána inicializační funkce. Uvnitř této funkce vloží generátor novou rovnici do tabulky rovnic a nastaví ji jako aktuálně generovanou. Následně mohou přicházet požadavky na generaci konkrétních prvků rovnice – konstant, proměnných, operátorů a vestavěných funkcí. Pro každý prvek je vytvořena struktura výpočtového prvku a je přidána na konec výpočtového předpisu aktuálně generované rovnice. Pro správné vyplnění všech částí výpočtového prvku je před jeho vložením ještě třeba provést další akce. Tyto akce se liší podle typu vkládaného prvku.

V případě zpracování proměnných je třeba proměnnou nejdříve vložit do podtabulky vstupních parametrů v tabulce zdrojů. Zdroj, ke kterému má být proměnná přidána, rozpozná generátor podle prefixu. Poté již můžeme ve výpočtovém prvku vyplnit odkaz na nově vložený záznam (*iid* zdroje – *iid* vstupního parametru). Pokud se tato proměnná již někdy dříve vyskytovala ve zdrojovém kódu, v tabulce vstupních parametrů je již vložena, a tak stačí zjistit její *iid*. Jak při výběru správného zdroje, tak následně při kontrole, zda již proměnná do příslušné tabulky parametrů nebyla vložena dříve, se využívá asociativního vyhledávání. Proto mají obě tyto tabulky index nad jmenným sloupcem.

V případě funkcí a operátorů musíme znát kód a počet parametrů prováděné operace. Následně ještě musíme zkontrolovat, zda počet parametrů ve zdrojovém kódu odpovídá počtu parametrů, které operace vyžaduje. Všechny operátory a vestavěné funkce mají uložený svůj prototyp ve statickém poli uvnitř generátoru. Prototyp obsahuje minimální a maximální počet vstupních parametrů (nutné v případě funkcí umožňujících proměnný počet parametrů), počet výstupních parametrů a u vestavěných funkcí i jméno. Prototypy jsou v poli seřazeny tak, aby jejich pořadí odpovídalo kódu operace, kterou reprezentují. U operátorů kód operace již známe, u funkcí ho podle jména musíme nejdříve vyhledat v poli prototypů. Počet parametrů vyžadovaný operátory je stanoven přímo gramatikou. Pokud je ve zdrojovém kódu použit počet chybný, není výraz syntaktickým analyzátorem vůbec přijat. Počet operandů tedy není nutné explicitně kontrolovat. U funkcí syntaktický analyzátor počet parametrů počítá. Tento počet musí ležet v rozsahu určeném příslušným prototypem.

Po vygenerování výpočtového předpisu je volána funkce pro každý výstupní parametr rovnice. Opět je zde nutné znát odkaz do podtabulky výstupních parametrů v tabulce zdrojů. Postup je stejný jako u parametrů vstupních.

Generace rovnice končí ve chvíli, kdy syntaktický analyzátor zavolá ukončující funkci. Generátor zneplatní odkaz na aktuálně generovanou rovnici a poslednímu prvku výpočtového předpisu v rovnici nastaví příznak, pomocí kterého interpret snadno rozpozná konec rovnice.

6.6.5 Kontrola existence proměnných

U vstupních i výstupních parametrů rovnice je prováděna kontrola, zda v dané databázové tabulce existují. Při generaci ovšem nejsou databázové tabulky součástí stejného procesu jako je tomu při interpretaci, a tak by bylo nutné přistupovat k nim vzdáleně. Běžně bývají v dopočtových souborech zpřístupněny dvě až tři databázové tabulky v podobě zdrojů, ale stovky až tisíce veličin v podobě proměnných. Vzdálené kontrolování existence proměnných by tak bylo velmi neefektivní, protože by vyžadovalo použití síťové komunikace (*cmi* využívá TCP/IP) pro každou prozatím netestovanou proměnnou. Proto se využívá přístupu jiného. Při výskytu první proměnné z daného zdroje je celá databázová tabulka přenesena do procesu překladače. Přenáší se pouze sloupec obsahující název veličiny, protože jiné sloupce nejsou pro kontrolu podstatné a zbytečně by přenos zpomalovaly. Kontrola je následně prováděna již v této lokální tabulce. Pro uložení přenesených tabulek slouží v *headru* vnořená tabulka lokalizovaných tabulek. Ze stejného důvodu jako u kontroly zdroje i zde překlad proběhne v pořádku i v případě neexistence požadované proměnné.

6.6.6 Zpracování filtru

Zpracování textového filtru s využitím rozhraní *dbi* probíhá nad lokalizovanými tabulkami. Na tabulku je aplikován požadovaný filtr a všechny prvky, které filtru odpovídají, jsou přidány do aktuální generované rovnice stejným postupem jako běžné vstupní proměnné. Počet takto vytvořených vstupních proměnných je nutné spočítat a uložit. Funkce, která má s textovým filtrem pracovat, si následně musí o uloženou hodnotu upravit počet vstupních parametrů, který obdržela od syntaktického analyzátoru.

6.6.7 Ukončení a uložení binárního souboru

Před závěrečným uložením je třeba provést ještě jeden krok. U podtabulek vstupních parametrů všech zdrojů správně vyplnit, do kterých rovnic tento parametr vstupuje.

Při interpretaci binárního kódu je kromě odkazu z výpočtového předpisu do tabulky vstupních parametrů nutný i odkaz zpětný. Jedná se tedy o seznam odkazů z prvku tabulky vstupních proměnných do všech rovnic, kde se proměnná vyskytuje jako vstupní parametr. Tento seznam odkazů je nutný pro správnou funkčnost změnových dopočtů. Ve chvíli detekce změny veličiny, nalezne interpret odpovídající záznam v tabulce vstupních parametrů. Díky seznamu zpětných odkazů lze snadno zjistit, které rovnice mají být vlivem změny přepočítány. Důvod, proč zpětný odkaz není vyplněn ve stejnou chvíli jako odkaz z rovnice do tabulky, je čistě optimalizační.

V dopočtech existuje vestavěná funkce, pomocí které je možné označit vstupní parametr rovnice jako takzvaný *pasivní*. Pasivní vstupní parametry nezpůsobují přepočet rovnice ve změnových dopočtech. Detekce změn a následný přepočet ve změnových dopočtech jsou právě nejkritičtější oblastí z hlediska výkonnosti interpretace, a proto je každá optimalizace v této části velmi žádoucí.

Implementace této funkce je díky opožděnému vyplňování odkazů přenesena již do generační fáze. Ve chvíli generace této funkce je z výpočtového předpisu aktuálně generované rovnice vyzvednut poslední prvek. Pokud je tento prvek proměnná, je této proměnné nastaven příznak pasivity. Jakýkoli jiný prvek než proměnná znamená chybu při použití funkce.

Doplňování seznamu odkazů do tabulek vstupních parametrů pak postupně prochází výpočtové předpisy všech rovnic a kdykoli narazí na proměnnou bez příznaku pasivity, zapíše odkaz na ni do příslušné tabulky vstupních parametrů.

6.7 Interpret

Nejdříve shrneme pár již známých faktů o interpretu. Interpret je implementován jako knihovna, která poskytuje ostatním programům rozhraní pro práci s binárním dopočtovým souborem. Slouží k zavedení dopočtu do běžícího procesu, provádění výpočtů popsaných binárním souborem a následně odinstalaci dopočtu. Dopočet tak může probíhat jako součást procesu, který je vlastníkem databázových tabulek v dopočtu využívaných, a tím odpadá nutnost využití meziprocesové komunikace při přístupu k databázovým datům. Všechny funkce rozhraní interpretu musí být reentrantní, protože jeden proces může pracovat s více dopočtovými soubory najednou. Nyní postupně rozebereme průběh interpretace a implementační triky, které využívá.

6.7.1 Zavedení dopočtu

Binární soubor je s využitím rozhraní *dbi* načten do paměti a zpřístupněn jako struktura popisující *header* tabulku. Poté je zkontrolováno, zda verze dopočtového souboru odpovídá verzi interpretu a jsou nastaveny globální příznaky podle příznaků instalačních jako například typ dopočtu. Jakmile je dokončena inicializace struktury headru, následuje příprava zdrojů.

Každý ze zdrojů musí být nejdříve zkontrolován. Probíhají stejné kontroly jako při generaci binárního souboru. V systému musí existovat všechny databázové tabulky s požadovanými sloupci a v nich všechny veličiny uložené v podtabulkách vstupních a výstupních proměnných. Proměnné neexistující v databázových tabulkách s nastaveným příznakem pro vložení, jsou vloženy nyní. Případně jsou vloženy všechny neexistující proměnné, pokud je instalační funkci předán příznak povolující toto chování. U kontroly tabulky výstupních proměnných se k jednotlivým prvkům ukládají *iid* odpovídajícího prvku v databázové tabulce, aby při ukončení výpočtu rovnice bylo možné výslednou hodnotu zapsat na správné místo do databáze. Při kontrole vstupních proměnných probíhá operace složitější. Jedná se o nejdůležitější optimalizační krok vedoucí ke zrychlení interpretace. Tomuto kroku budou věnovány následující dvě podkapitoly. Po dokončení kontroly existence proměnných registruje interpret u zdrojem zpřístupněné databázové tabulky callbackovou funkci. Jako uživatelská data, která budou callbackové funkci předána, přitom vloží aktuálně zpracovávaný zdroj.

Tato callbacková funkce je dále využívána k více účelům. Slouží k počáteční inicializaci hodnot proměnných podle veličin, odchytávání změn hodnot veličin a kontroluje, zda veličiny smazané v databázi v průběhu dopočtu nejsou vstupními proměnnými.

6.7.2 Princip filtrování v *dbi* tabulkách

Pro pochopení zmiňovaného optimalizačního kroku je nutné nejdříve představit další možnosti filtrování a vysvětlit princip, na kterém filtrování záznamů v *dbi* tabulkách pracuje.

Dbi rozhraní nabízí kromě již známého textového filtru i jiné možnosti filtrování obsahu tabulek. Jednou z nich je filtrování na základě poskytnutého seznamu *iid*. Filtru pak odpovídají všechny záznamy, jejichž *iid* bylo předáno v seznamu. Implementace optimalizace právě tento druh filtru využívá.

Filtr se vždy týká kurzoru, přes který se přistupuje k tabulce, nikoli tabulky samotné. Jak již bylo zmíněno, pro jednu tabulku může existovat libovolný počet kurzorů, a tedy i libovolný počet různých filtrů. Všechny druhy filtrů pracují na stejném principu. Ve chvíli nastavení filtru je mezi funkce rozhraní pro manipulaci s tabulkou a tabulku samotnou

vložena filtrovací vrstva. Tato vrstva je interně vytvořená nová tabulka, která obsahuje pouze prvky odpovídající filtru a společně s nimi mapování prvků na příslušné prvky skutečné tabulky, nad níž byl filtr vytvořen. Kurzor, kterým byla ovládána tabulka před filtrováním, je nyní nastaven na interní tabulku obsahující pouze filtrované prvky. Filtrovací vrstva je tak plně transparentní. Interní tabulka vzniklá za účelem filtrování má jednu speciální vlastnost týkající se callbackové funkce. V případě volání registrované callbackové funkce umísťuje do struktury specifikující událost také pořadí prvku ve filtrované tabulce. K tomuto účelu využívá pole struktury zvané uživatelská identifikace.

6.7.3 Filtrování vstupních proměnných v databázových tabulkách

Nyní už můžeme přejít k implementaci. Vychází se z předpokladu, že podtabulka vstupních proměnných daného zdroje obsahuje podmnožinu prvků databázové tabulky zdrojem zpřístupněné. Tento předpoklad by nebyl splněn pouze v případě, že by tabulka vstupních proměnných obsahovala proměnné v databázi neexistující, což je považováno za chybu. Kontrola existence proměnných ze vstupní tabulky musí probíhat pomocí asociativního vyhledávání v příslušné tabulce databázové. V případě nalezení či vložení proměnné do databázové tabulky veličin je uloženo *iid* nalezeného či vloženého záznamu. Následně je nad kurzorem zpřístupňujícím databázovou tabulku nastaven filtr se seznamem všech uložených *iid*. Výsledná filtrovaná tabulka tak obsahuje stejné prvky jako tabulka vstupních proměnných právě zpracovávaného zdroje. Callbacková funkce je následně registrována již tabulce filtrované. Do callbackové funkce tak přichází pouze informace týkající se vstupních proměnných, nikoli ostatních prvků databázové tabulky.

Jednotlivé *iid* prvků byly do seznamu použitého pro filtr vkládány v pořadí, v jakém jsou prvky uloženy v příslušné tabulce vstupních proměnných. Díky tomu jsou pak v tomto pořadí uloženy i prvky v interní filtrované tabulce představující tabulku databázovou. Tabulka vstupních proměnných daného zdroje a interní filtrovaná tabulka mají tedy shodné prvky na shodných pozicích. Ve chvíli volání callbackové funkce je tak předána ve struktuře popisující událost absolutní pozice prvku v tabulce vstupních proměnných a ve struktuře obsahující uživatelská data je předán zdroj, který danou podtabulku vstupních proměnných vlastní. Při příchodu události inicializace, modifikace či smazání prvku do callbackové funkce stačí pouze vyhledat prvek na předané absolutní pozici v příslušné tabulce vstupních proměnných a ihned je možné provést požadovanou akci.

Pro nalezení požadovaného prvku tedy stačí provést operaci přístupu na absolutní pozici v tabulce. Tato operace je téměř stejně rychlá jako nejrychlejší možný přístup k prvkům přes *iid*. Naproti tomu v případě nefiltrovaných vstupních tabulek by do callbackové funkce přicházely všechny změny z celé databázové tabulky. Bylo by tak nutné provést asociativní vyhledávání, zda změněný prvek vůbec patří mezi sledované. Asociativní vyhledávání je operací podstatně náročnější než operace přístupu na absolutní pozici, ale hlavně by bylo prováděno pro každou změnu v databázové tabulce a to i v případě, že na tuto změnu vůbec nebylo třeba reagovat. Databázové tabulky veličin přitom obsahují desetitisíce prvků.

6.7.4 Úvodní přepočet celého souboru

Změnové i periodické dopočty mají společný úvodní přepočet všech rovnic ihned po zavedení. Před provedením celkového výpočtu je ale nutné nejdříve inicializovat hodnoty proměnných podle hodnot veličin v databázi. Musíme si uvědomit, že ve výpočtových předpisech pro počítání rovnic vystupují proměnné jako odkazy do podtabulek vstupních proměnných uložených v tabulkách zdrojů, nikoli přímo do tabulek databázových. Proto je

třeba nejdříve záznamy o proměnných v tabulkách vstupních proměnných naplnit skutečnými hodnotami veličin. Tento problém lze vyřešit přímo funkcí rozhraní *dbi*. Zažádáme databázovou tabulku (přesněji její filtrovanou verzi) o takzvaný *globální dotaz*. Následně bude pro každý prvek, který žádaná tabulka obsahuje, zavolána registrovaná callbacková funkce a prvek bude předán jako součást struktury specifikující událost. Interpret tedy musí zažádat o globální dotaz každou databázovou tabulku, jejíž veličiny vystupují v dopočtu jako vstupní proměnné.

Důležité je, které složky vyžádaných databázových prvků (hodnota, kvalita, alarm, čas) mají být do tabulky vstupních proměnných uloženy. To se pozná podle formátu, který byl uživatelem nastaven v definici zdroje zpřístupňujícího danou databázovou tabulku. Složky, které nemají ve výpočtu pro daný zdroj vystupovat, jsou nastaveny na neutrální hodnotu. Neutrální hodnotou je myšlena taková hodnota, která počítání s danou složkou nijak neovlivní. Neutrální hodnotu snadno zjistíme podle pravidel pro počítání jednotlivých složek z kapitoly 4.3.4:

složka	neutrální hodnota
hodnota	neexistuje
kvalita	implicitní kvalita dopočtového souboru
alarm	vynulovaný příznak aktivity a nulový stupeň závažnosti
čas	čas 0 = Unix epoch time (půlnoc 1.1.1970)

Neutrální hodnota pro hodnotu veličiny samozřejmě neexistuje (neexistuje číslo, které by bylo ke všem matematickým operacím prováděným v dopočtech neutrální). To však není překážkou, protože formát musí hodnotu vždy obsahovat.

Po úspěšné inicializaci proměnných projde interpret postupně celou tabulku rovnic uloženou v *headru* a s využitím výpočtového předpisu rovnice provede.

6.7.5 Provádění rovnic

Výpočet výrazů uložených v RPN běžně využívá zásobníkové implementace. Operandy jsou ukládány na zásobník. Ve chvíli provedení operace je ze zásobníku vyjmut počet operandů, který operace vyžaduje, operace je provedena a výsledek je uložen zpět na zásobník. Jakmile je výpočet výrazu ukončen, výsledek je na vrcholu zásobníku.

Výpočet rovnic v interpretu probíhá s využitím pole a indexu, který ukazuje za konec platných prvků v tomto poli. Výpočtový předpis rovnice uložený v RPN je postupně čten od začátku do konce. Pokud je přečten operand, je vložen do pole na místo, kam ukazuje index a index je následně inkrementován. Pokud je z výpočtového předpisu přečten operátor, dekrementuje se index o počet operandů snížený o jedna. Následně se operace provede (operandy pro operaci jsou na patřičných pozicích za indexem platných prvků) a před místo, kam ukazuje index je zapsán výsledek. Algoritmus je vlastně shodný se zásobníkovou implementací popsanou výše, pouze optimalizovaný pro rychlost. Ke všem prvkům pole lze přistupovat přímo, na rozdíl od zásobníku, kde lze přistupovat pouze k prvku poslednímu. Ve výpočtovém poli jsou tak prvky přepisovány přímo přes sebe a tím se ušetří operace vyjmutí ze zásobníku a následného vložení výsledku. Pole sloužící pro výpočet rovnice bude dále označováno jako výpočtový buffer.

Jak již bylo vysvětleno, prvky výpočtového předpisu pro proměnné obsahují pouze odkaz do tabulky vstupních proměnných. U výpočtového bufferu je nutné mít tyto informace uloženy přímo v jednotlivých prvcích pole. Proto jsou ve chvíli ukládání proměnné

do výpočtového bufferu zjištěny informace z tabulky vstupních proměnných a do bufferu nakopírovány.

Prvek výpočtového bufferu obsahuje kromě hodnoty, kvality, alarmu a času ještě masku, ze které se při zápisu výsledků do databáze zjišťuje, které složky mají být zapsány. Maska se přenáší při výpočtu mezi jednotlivými prvky bufferu pomocí operace bitového součinu a lze ji modifikovat vestavěnými funkcemi jako je například již vícekrát zmiňovaná *calc_mask* nebo *_I* – nezapisovat hodnotu při neaktuální kvalitě.

6.7.6 Výpočet závislý na kvalitě

Z pravidel pro výpočet jednotlivých složek proměnných popsaných v kapitole 4.3.4 vyplývá, že výpočet výsledné hodnoty rovnice je závislý nejen na hodnotě, ale i na kvalitě jednotlivých vstupních proměnných. Zásadní je přitom rozlišovat kvalitu chybnou. Chybná kvalita v dopočtech značí chybu a prvek s touto kvalitou je z rovnice eliminován. V případě, že daný výpočet eliminaci prvku neumožňuje, se chyba šíří dál.

Před provedením operace musí být zkontrolováno, zda nemají operandy kvalitu chybnou. Pokud má u binární aritmetické či logické operace jeden z operandů kvalitu chybnou, operace není vůbec provedena a jako výsledek je přímo nastaven operand druhý. U operací porovnávacích a bitových je pouze do pole pro výsledek nastavena chybná kvalita. Funkce pro řízení toku pak takovéto podmínky vůbec nevyhodnocují a předají dál kvalitu chybnou. Stejně se v případě vstupní veličiny s kvalitou chybnou chová i většina dalších vestavěných funkcí. U unárních operací není nutné kvalitu kontrolovat. Kvalita se totiž unární operací změnit nemůže a kontrola kvality probíhá i při zápisu výsledků rovnic do databáze. Chybnou kvalitu výsledku unární operace tak odhalí buď následující binární operace či vestavěná funkce a nebo přímo zápis výsledku.

6.7.7 Zápis výsledků do databázové tabulky

Ve chvíli ukončení výpočtu rovnice zůstane výsledek na začátku výpočtového bufferu, případně může v bufferu zůstat výsledků více, pokud má rovnice více výstupních parametrů. Nyní je třeba výstupní parametry zapsat do odpovídající databázové tabulky. Seznam výstupních proměnných je uložen u každé rovnice v podobě odkazů do tabulek výstupních proměnných. Pořadí odkazů na výstupní proměnné uložené v poli u rovnice odpovídá pořadí výstupních parametrů, které zůstaly ve výpočtovém bufferu po dokončení výpočtu rovnice. Pro každou zapisovanou proměnnou je třeba provést následující akce:

- Zjistit, zda kvalita zapisovaného výsledku není chybná. Pokud ano, zápis se neprovede.
- Pomocí odkazu v seznamu výstupních proměnných zjistit záznam v tabulce výstupních proměnných.
- Ze záznamu v tabulce výstupních proměnných zjistit odpovídající prvek v databázové tabulce. Za tímto účelem má záznam v tabulce výstupních proměnných uloženo *iid* odpovídajícího databázového prvku.
- Vyhodnotit masku zapisovaných složek uloženou u výsledku v prvku výpočtového bufferu a podle ní sestavit zapisovaný prvek.
- Při sestavování musí být brán ohled na to, které složky z databázové tabulky byly nastaveny ve formátu při definice zdroje. Nenastavené složky se nezapisují.

- Sestavený prvek zapsat do databázové tabulky s využitím modifikační funkce rozhraní *dbi*.

Všimněme si, že teprve při zápisu výsledku bylo poprvé za celý výpočet rovnice řešeno, s kterými složkami se má pracovat. Jednak by bylo zbytečně složité při výpočtu kontrolovat, s kterými složkami daná proměnná počítá, ale především s kvalitou je třeba počítat vždy, aby bylo dosaženo správného šíření chyby u vestavěných funkcí.

Důležité je zápis do databázové tabulky provádět s využitím modifikační funkce. Interpretace dopočtů probíhá jako součást procesu *Rdb*, a tak by bylo možné hodnoty zapisovat s využitím přímého přístupu. V tomto případě by ale nebyla volána callbacková funkce hlásící modifikaci zapisovaného prvku, což by mělo fatální důsledek nejen pro změnové dopočty.

6.7.8 Aktivace změnového dopočtu a cyklické závislosti

Oba druhy dopočtů potřebují odchytávat změny veličin v databázi a upravovat tak záznamy v příslušných tabulkách vstupních proměnných. Pokud se navíc jedná o dopočet změnový, po aktualizaci záznamu změněné vstupní proměnné dochází k aktivaci výpočtu. Záznam v tabulce vstupních proměnných obsahuje seznam rovnic, které je třeba přepočítat. Pro každou rovnici je tedy postupně spuštěn výpočet popsaný v kapitole 6.7.5. Jakmile je rovnice spočtena, přejde se na zápis výsledků do databáze. Při zápisu výsledků je použita modifikační funkce a přitom je modifikovaná databázová tabulka lokální (interpret běží jako součást *Rdb*). Jak už bylo vysvětleno v kapitole 3.2.5 v případě lokální tabulky je volána callbacková funkce ještě než řízení toku programu opustí modifikační funkci. Pokud je tedy modifikovaná veličina vstupní proměnnou jiné nebo i právě počítané rovnice, je opět volána registrovaná callbacková funkce, která znovu spustí stejnou posloupnost kroků popsanou výše. Zde jasně vidíme, že případné cyklické závislosti rovnic by vedly k nekonečné rekurzi, přetečení zásobníku a následnému pádu programu. Řešením by mohlo být zapisování změn formou přímé modifikace databáze, čímž se vyhneme rekurzi. V tomto případě by bylo nutné hlídat změny způsobené přepočtem jiným netriviálním způsobem.

Řešení cyklických závislostí rovnic při změnových dopočtech je ale právě díky rekurzivnímu způsobu zpracování změn velmi snadné. Každá rovnice obsahuje kromě výpočtového předpisu a seznamu výstupních proměnných ještě sadu příznaků. Jeden z těchto příznaků značí, že je rovnice právě prováděna. Tento příznak je otestován a následně nastaven před začátkem zpracování rovnice a zrušen po dokončení zápisu výsledků do databáze. Tedy až ve chvíli, kdy už skončily všechny callbackové funkce rekurzivně vyvolané modifikací databáze. Pokud rovnice ještě nedokončila zápis výsledků a má být vlivem rekurzivního zpracování počítána znovu, test příznaku vyjde pozitivní a výpočet rovnice vůbec nezačne.

Rekurzivní řešení výpočtu s sebou ale přináší i další problém, který se projeví ve chvíli, kdy má rovnice více výstupních proměnných. Modifikace databáze musí postupně proběhnout pro všechny výstupní proměnné, které jsou uloženy ve výpočtovém bufferu. Další rovnice, spuštěná zápisem první výstupní proměnné do databáze, však použije pro výpočet a uložení výsledků opět výpočtový buffer a přepíše tak zbývající výsledky rovnice předcházející. Tento problém byl vyřešen tak, že v případě více výstupních proměnných je výpočtový buffer obsahující výsledky duplikován. Díky tomu počítá změnou vyvolaná rovnice s bufferem jiným, a nedojde tak ke ztrátě předchozích výsledků, jež doposud nebyly zapsány. Duplikát výpočtového bufferu je pak uvolněn až po zápisu všech výstupních proměnných.

6.7.9 Implementace složitých vestavěných funkcí

Dopočtový jazyk obsahuje složité vestavěné funkce, jako jsou konečné a okamžité integrály a průměry počítané v periodických dopočtech, či specializované funkce jako již několikrát zmiňovaný *split* nebo *trip*. Implementaci těchto funkcí přenesli jejich autoři z původní verze interpretu a následně optimalizovali díky výpočtovému bufferu, který umožňuje přímý přístup ke všem svým prvkům.

Kapitola 7

Shrnutí a budoucí vývoj

Kapitola shrnuje implementovaná rozšíření a jejich hodnocení uživateli. Dále je představeno průběžné testování výkonnosti interpretu. Na závěr se věnuje budoucímu vývoji projektu a možným vylepšením.

7.1 Zhodnocení splnění uživatelských požadavků

S pomocí nově zavedeného preprocesoru se podařilo vyřešit požadavky na definice uživatelských funkcí a využívání funkcí již jednou definovaných. Uživatelské funkce byly nahrazeny textovými makry. Preprocesor umožňuje vkládání externích souborů, a tak mohou být makra definována v externích souborech a do souboru s dopočtovými rovnice následně vložena.

Jak se ale ukázalo, nevýhody textových maker ve srovnání s funkcemi uživatelům nevadí. Podle jejich popisu by parametry funkcí byly předávány vždy pouze odkazem, aby byla možná jejich přímá modifikace. Toto chování je u maker díky textové náhradě parametrů implicitní. Naopak makra do jazyka přinesla jednu zásadní výhodu. Uvnitř těla totiž umožňují řetězení parametrů s textovými řetězci, což je u některých typů dopočtů velice přínosné. Obvyklý tvar názvů veličin byl představen již v kapitole 3.2.2. Veličiny týkající se jednoho energetického uzlu se často liší pouze poslední částí jména, například `AASC2:V203BA:U`, `AASC2:V203BA:U_est` a podobně. Jedním z typických využití dopočtů je počítání sum a nulových bilancí výkonu v uzlech sítě. Tento výpočet je pro uzly stejného charakteru shodný, liší se pouze úvodními částmi jména specifikujícími síť a uzel. Díky řetězení parametrů je jednoduché tento výpočet popsat makrem, toto makro poté volat s řetězcem specifikujícím uzel a konkrétní veličiny tvořit uvnitř marka řetězením s příslušnou poslední částí jména.

Možnost definice pomocných proměnných byla vyřešena přidáním pomocných zdrojů. Uživatelé se již dříve snažili problém absence pomocných proměnných řešit pomocí operátoru @ a vkládat tak nové pomocné veličiny do databáze. To mělo samozřejmě nežádoucí následky. Díky tomu bylo ale zvolené řešení pro uživatele zcela intuitivní. Zpřístupnění pomocné databázové tabulky v podobě definice zdroje tak zůstalo na uživatelích. Ti si díky tomu mohou sami zvolit jméno zdroje, přes které budou proměnné přístupné. Tabulka pomocných proměnných, která je uložena přímo jako součást *headru*, je na přání uživatelů zpřístupněna implicitně pod prázdným jménem zdroje. Proměnné z této tabulky jsou v programu dostupné přidáním prefixu *:* před jméno.

Prozatím jediný požadavek, jehož řešení uživatele příliš neuspokojilo, je zadávání pro-

měnných s pomocí filtru. Prvním problémem je, že řetězec popisující filtr je přímo předáván *dbi* funkci pro zpracování textového filtru. Tím je možnost používání omezena pouze na veličiny z jedné tabulky. Jména veličin ve filtru navíc nesmí obsahovat prefix určující zdroj, protože filtrovací funkce samozřejmě neví, že tento prefix do hledaného jména nemá zahrnout, a tak by byl filtr špatně vyhodnocen. Zadávání výčtu veličin filtrem je tak v současnosti možné pouze z jednoho zdroje a to toho, který je ve chvíli použití filtru nastaven jako implicitní. Tento problém lze vyřešit rozdělením a úpravou řetězce filtru na více řetězců podle zdrojů u jednotlivých prvků ve filtru vystupujících. Tyto rozdělené řetězce následně předat funkcím pro textovou filtraci nad příslušnými tabulkami. Vzhledem k syntaktickým možnostem filtru se jedná o problém poměrně složitý, důležité ale je, že řešitelný. Druhý problém v současné situaci prozatím řešení nemá. Jde o to, že filtr je vyhodnocen již při překladu. Velmi nepříjemný je ale fakt, že toto chování přímo odporuje dřívější koncepci práce s jazykem popsané v kapitole 4.1. Při používání filtru je naopak překlad závislý na tom, kdy proběhne interpretace a je třeba jej provádět až před zavedením dopočtu. Textový filtr by jinak mohl být vyhodnocen chybně.

7.2 Předběžné testování výkonnosti interpretu

Nejdůležitějším požadavkem byla rychlostní optimalizace interpretu. Jednalo se o požadavek nutný, protože v případě dalšího rozšiřování rozsáhlých změnových dopočtových souborů, jakým je například dopočet regulátoru salda, by jazyk nebyl schopen plnit účel, pro který byl vyvinut.

Prozatím uskutečněné testování je pouze předběžné, protože interpret je především z důvodu implementace blokovacích podmínek stále ve vývoji. Test výkonnosti využívá simulace změn databázových veličin. Za účelem simulování změn byl pracovníky firmy Elektrosystem poskytnut program, který umožňuje generovat přicházející změny do *Rdb*. Tyto generované změny jsou ekvivalentní změnám, které přicházejí například z telemetrických měření. Kdykoli program odešle změnu, je blokován, dokud *Rdb* tuto změnu nevyhodnotí. *Rdb* následně informuje program, že proběhlo zpracování změny, a ten může pokračovat v simulaci změn dalších. Nyní už přejdeme k průběhu samotného testu.

V databázi je vytvořena nová testovací veličina. Při vyhodnocení její změny tedy *Rdb* nemusí provádět žádné jiné akce než je zápis této změněné hodnoty do databáze. Simulační program je nastaven tak, aby generoval jistý počet změn této veličiny. Přitom je zaznamenávána doba běhu tohoto programu. Následně je do *Rdb* nainstalován změnový dopočtový soubor, který obsahuje rovnici s danou testovanou veličinou jako vstupním parametrem. V *Rdb* se nyní jako součást vyhodnocení změny veličiny musí spustit i nainstalovaný dopočet. Opět je měřen čas běhu simulačního programu. Poté se spočte rozdíl délky běhu simulačního programu s nainstalovaným dopočtem a bez něj. Výsledek je čas, který potřebuje interpret pro výpočet testovací rovnice.

Výsledek tohoto testu ukazuje, že nový interpretu je zhruba o 50 % rychlejší než jeho předchůdce. Test ovšem ověřuje pouze čistý výpočet rovnice. Tohoto zrychlení bylo tedy dosaženo pouze při vyhodnocování rovnice, což je považováno za úspěch. Výraznější zrychlení se projeví až při v skutečném běhu *Rdb*, kdy bude přicházet několikanásobně větší množství změn. Zásadní rychlostní optimalizací je totiž filtrování vstupních proměnných. Zajišťuje, aby jednotlivým nainstalovaným dopočtům přicházely pouze pro ně významné změny, což snižuje režii dopočtů při každém příchodu změny do *Rdb*.

Výše popsaný test je prováděn průběžně vždy po dokončení ucelenějšího úseku. Pokud tedy dojde ke ztrátě výkonnosti, je tak snadno zjistitelné, které úpravy jsou toho příčinou.

7.3 Budoucí vývoj

Před implementací vylepšení a dalších optimalizací je nutné nejdříve dokončit stávající rozpracované požadavky. Hlavním požadavkem jsou již zmíněné blokovací podmínky.

Blokovací podmínky jsou „specialitou“ systému *RIS*, umožňující uživateli definovat podmínky, které musí být splněny, aby mohla být vykonána požadovaná akce. Používají se například pro povolení dálkového ovládání veličiny. Jednou z možností při dálkovém ovládání veličiny je vzdáleně zadaná změna hodnoty veličiny přímo v *Rdb*. Správce měřičů dodávajících hodnotu veličin může využít následujících dvou blokovacích podmínek pro zamezení dálkového ovládání: měřič nedodává hodnotu nebo je veličina ve fixním módu. Po aplikaci těchto podmínek je možné ručně měnit hodnotu veličiny pouze pokud je měřič odpojen nebo veličina nastavena do fixního módu. Správce tak znemožní dohlížejícím dispečerům měnit hodnotu dodávané veličiny. Situace je ve skutečnost mnohem složitější. Předtím než je umožněno přímo zadat blokovací podmínky, kontrolují se různá oprávnění a mnoho dalších věcí. Pro detailnější vysvětlení této problematiky by bylo třeba nejméně jedné celé kapitoly. Zde tedy pouze uvedeme, že se jedná o požadavek velmi náročný, jehož splnění při zachování dostatečné rychlosti interpretace zabere několik dalších týdnů. Dalším nutným požadavkem je zobecnění implementace zadávání výčtu proměnných pomocí textového filtru. Tento problém byl již diskutován v předcházející kapitole.

Po dokončení těchto zásadních požadavků nastane fáze příprav pro nahrazení stávajícího překladače a interpretu novými. V této fázi bude velmi důležité rozsáhlé a kvalitní testování. Vzhledem k tomu, že byla zachována 100% zpětná kompatibilita se stávající verzí jazyka, se jako nejvhodnější v současnosti jeví testování využívající srovnání. Stejný dopočtový soubor přeložit oběma verzemi překladačů a výsledné binární soubory nechat interpretovat opět oba interprety samozřejmě při shodném obsahu databáze. Pro tento účel bude nutné vyvinout automatický testovací nástroj a s jeho pomocí tento test provést s co nejvíce aktuálně využívanými dopočtovými soubory.

7.4 Plánovaná vylepšení

Vyvíjený překladač a interpret samozřejmě obsahuje nedostatky, které by bylo vhodné před nasazením do praxe odstranit. Z časových důvodů to ale nejspíše již nebude možné. K odstranění těchto nedostatků dojde až v další vývojové fázi. Především je to velmi nepropracované podávání chybových hlášení v případě syntaktických chyb. To ale pro uživatele není žádnou katastrofou, protože dosavadní syntaktický analyzátor díky průběžným úpravám v chybových hlášeních často neudával ani správné číslo řádku. Kromě tohoto závažnějšího nedostatku obsahuje jazyk více nedostatků drobnějších. Nešikovně je implementováno například vyhledávání prototypů funkcí ve statickém poli prototypů v generátoru binárního kódu. Jedná se o sekvenční vyhledávání na shodu řetězce (jména funkce). Mnohem vhodnější by bylo použít vyhledávání asociativní. Množina řetězců je konstantní, a tak by bylo možné efektivní mapovací funkci snadno vytvořit s využitím automatického nástroje jako je například *gperf* (viz. [8]).

Užitečné by byly i další rychlostní optimalizace interpretace. Usnadnit práci interpretu je možné i generací efektivnějšího binárního kódu, konkrétně tedy výpočtových předpisů pro rovnice. Například zrychlení interpretace o 10 % na úkor dvakrát pomalejší generace binárního kódu je jistě žádoucí. Jeden přístup je přenášení operací z interpretu do generátoru. Tento způsob byl již využit při implementaci vestavěné funkce pro pasivaci vstupů rovnice. V rámci tohoto přístupu se nabízí implementovat vyhodnocování konstantních

výrazů v době překladu. Otázkou ale je, zda má tato optimalizace vůbec význam vzhledem k mizivému procentu výskytu takovýchto výrazů. Druhým zajímavějším způsobem je úprava výpočtového předpisu na efektivnější formu. Zástupcem této skupiny optimalizací může být například vytýkání před závorku. To je užitečné zejména díky tomu, že výrazy uložené v RPN závorky nepotřebují, a tak by úprava vedla ke snížení počtu prvků výpočtového předpisu. Nejlepší bude vysvětlení na příkladu. Výraz $3 * a + 3 * b$ zapsaný v infixové notaci lze snadno upravit na $3 * (a + b)$. Tuto úpravu lze samozřejmě provést i v RPN: $3\ a * 3\ b * + \rightarrow a\ b + 3\ *$. Díky vytýkání před závorku se tak ušetřily dva prvky výpočtového předpisu a jedna operace, což v případě často prováděné rovnice může přinést nezanedbatelné zrychlení.

Kapitola 8

Závěr

Cílem této bakalářské práce bylo kompletně rekonstruovat a rozšířit jazyk umožňující definici uživatelských dopočtových rovnic využívaný v řídícím systému *RIS*. Rozšíření implementovat s ohledem na požadavky uživatelů a zaměřit se přitom na efektivitu interpretace.

Při studiu využití jazyka v systému byl zjištěn nepříjemný konceptuální nedostatek. Tímto nedostatkem je úzká vazba jazyka na program pracující se statistikami, která velmi zkomplikovala úpravy. Za účelem vyřešení požadavků byl přidán textový preprocesor, který umožnil do jazyka začlenit rozšíření způsobem slučitelným se vzniklými omezeními.

Projekt byl rozdělen na čtyři samostatně testovatelné celky: preprocesor, lexikální a syntaktický analyzátor, generátor binárního kódu a interpret. Implementace všech částí využívá univerzálního tabulkového rozhraní systému *RIS*. Toto rozhraní bylo použito pro práci se základními datovými strukturami, při generování binárního kódu i pro spolupráci s databází systému. Lexikální a syntaktický analyzátor byly vygenerovány s využitím programů Flex a Bison. To umožňuje snadnou implementaci budoucích změn či rozšíření. Interpret byl podle svého původního vzoru navržen a implementován jako rozhraní umožňující provádět přeložené dopočtové rovnice přímo jako součást jiných procesů.

Při implementaci projektu bylo věnováno hlavní úsilí výkonnosti interpretu. Značnou optimalizaci oproti předcházející verzi zaznamenal interpret v oblasti vyhodnocování rovnic. Zásobníková implementace vyhodnocování rovnic uložených v RPN byla nahrazena výpočtovým polem, kde jsou průběžně vyhodnocované výrazy přepisovány přes sebe. Jsou tak ušetřeny operace vyjmutí a opětovného vložení výsledku do zásobníku. Nejdůležitějším optimalizačním krokem interpretu bylo zavedení filtrování vstupních veličin. Energetické veličiny vstupující do rovnic jsou uloženy v rozsáhlých databázových tabulkách. Původní verze interpretu snímala změny z celých tabulek a následně vyhodnocovala, zda se změny týkají počítaných rovnic či nikoli. Změna veličiny tak musela být vyhodnocena všemi aktivními dopočty. I nejrozsáhlejší dopočtové soubory přitom obsahují maximálně 10 % veličin uložených v databázi. Nyní jsou změny snímány pouze z filtrovaných verzí tabulek. Konkrétní dopočty tak obdrží pouze změny, které vyžadují. Díky tomu je možné efektivněji vyhledat rovnice, kterých se tyto změny týkají. Především ale toto chování výrazným způsobem snižuje celkovou režii dopočtů v systému zpracování změn veličin.

Díky přibývajícím požadavkům uživatelů je projekt stále ve vývoji. Nebylo proto doposud provedeno komplexnější testování. V předběžných testech se jeví nový interpret přibližně o polovinu výkonnější než jeho předchůdce. Avšak vzhledem k typu provedených optimalizací se při plném zatížení v praxi očekává zrychlení až několikanásobné.

Tato práce byla ve zúženém rozsahu prezentována na soutěži studentských prací Student EEICT 2010 a je publikována ve sborníku soutěže (viz. [1]).

Literatura

[1] *Proceedings of the 16th Conference STUDENT EEICT 2010*, Brno: NOVPRESS s.r.o, 2010, ISBN 978-80-214-4076-0.

[2] Elektrosystem a.s.:*Řídící a informační systém RIS* [online]. [cit. 2010-04-22]. URL http://www.esys.cz/ris.php

[3] Oracle:*Oracle Database* [online]. [cit. 2010-04-23]. URL http://www.oracle.com/us/products/database/index.html

[4] C++ Reference:*C munerics library* [online]. [cit. 2010-04-25]. URL http://www.cplusplus.com/reference/clibrary/cmath

[5] The flex manual:*Start Conditions* [online]. [cit. 2010-04-27]. URL http://flex.sourceforge.net/manual/Start-Conditions.html

[6] A GNU Manual:*A self-referential macro* [online]. [cit. 2010-04-27]. URL http://gcc.gnu.org/onlinedocs/cpp/Self_002dReferential-Macros.html

[7] A GNU Manual:*The C Proprocessor* [online]. [cit. 2010-04-27]. URL http://gcc.gnu.org/onlinedocs/cpp

[8] *Gperf* [online]. [cit. 2010-05-02]. URL http://www.gnu.org/software/gperf

[9] Aho, A. V.; Lam, M. S.; Sethi, R.; aj.: *Compilers: Principles, Techniques, and Tools*. Boston: Addison Wesley, 2007, ISBN 0-321-49169-6.

[10] Levine, J. R.: *flex & bison*. O'Reilly Media, Inc, 2009, ISBN 978-0-596-15597-1.

[11] Meduna, A.: *Automata and Languages: Theory and Application*. London: Springer, 2000, ISBN 1-85233-074-0.

[12] Mátl, A.: *Správcovská příručka systému Ris*. Elektrosystem a.s., 2009.

[13] Wikipedia: *LALR parser - Wikipedia, the free encyclopedia* [online]. [cit. 2010-04-28]. URL http://en.wikipedia.org/wiki/LALR_parser

[14] Wikipedia: *Reentrant - Wikipedia, the free encyclopedia* [online]. [cit. 2010-04-29]. URL http://en.wikipedia.org/wiki/Reentrant_(subroutine)

[15] Wikipedia: *Reverse Polish notation - Wikipedia, the free encyclopedia* [online]. [cit. 2010-04-29]. URL http://en.wikipedia.org/wiki/Reverse_Polish_notation

Dodatek A

Ukázka použití *dbi* a *cmi*

```
// definice prototypu tabulky, kterou zpracuje drcc na struktury
// nested je vnorena tabulka, ktera se definuje obdobnym zpusobem
%%
TABLE example
    name    VARCHAR    DBNAMEMAX
    year    INT        int
    time    DATETIME
    nested  CURSOR
INDEX name UNIQUE name
INDEX year         year
%%
// uzitecne makro definujici staticke promenne podle prototypu
db_def_tab(example);
// vyvoreni nove pametove tabulky podle prototypu
// example_tab je staticka promenna definovana makrem vyse
dbdc_t cursor = DB_create("example_table", &example_tab, 0);
// naplneni tabulky daty
// example_t je prototyp prvku tabulky definovany drcc
for(int i = 0; i < 100; i++) {
    example_t exam = {0};
    sprintf(tmp, "name%d", i);
    exam.name = tmp;
    exam.year = i;
    DB_insert(cursor, &exam);
}
// serazeni podle roku, konstantu TOR_TEST_YEAR definuje drcc
DB_order(cursor, TOR_TEST_YEAR);
// vyzvednuti zaznamu s nejnizsim rokem
example_t *exam = DB_first(cursor);
// registrace procesu jako cmi serveru spojena
// se zverejnenim vsech lokalnich tabulek
DB_listen("/exam_srv");
// spusteni smycky udalosti
CM_execute();
```

Dodatek B

Ukázka zdrojového kódu dopočtů

Krátký upravený úsek zdrojového kódu dopočtů pro řízení regulace salda České republiky. Před podobnějším procházením příkladu doporučujeme nejdříve práci přečíst (minimálně kapitolu 4).

Ukázka obsahuje definice zdrojů pro zpřístupnění dvou hlavních databázových tabulek veličin, nastavení implicitního zdroje (proměnné z tohoto zdroje není dále nutné prefixovat jménem zdroje) a dvě konkrétní dopočtové rovnice.

První rovnice popisuje situaci, kdy je hodnota veličiny `gAGC:SL:P` závislá na veličině `gAGC:SLM:P`. Pokud není první veličina aktuální nemá být hodnota druhé veličiny uvažována.

Druhá rovnice je složitější. Popisuje měření podílející se na saldu. Konkrétně se jedná o dopočet ztrát energie způsobeným situací, kdy příslušný měřič neměří přímo v místě průtoku energie. Pro zpřehlednění složité rovnice bylo využito textového makra, jehož definici ale původní dopočty neumožňují.

```
// energeticky vztah pro vypocet ztrat na linkach
// kde je predavaci bod mereni lezi jinde nez meric
#define ztraty(l, r, b, u) \
    (r * (sqr(l:#P) + sqr(l:#Q - sqr(u) * b * 0.0000005)) / sqr(u))

// definice zdroje A pro zpristupneni databazove tabulky
// 'analog' - nazev databazove tabulky bez prefixu procesu
// name - sloupec obsahujici jmena veliciny
// 'value quality alarm' - format zpristupneni = slozky pouzite ve vypoctu
// podle ! ukoncujiciho format je zdroj permanentni
source A = 'analog', name, 'value quality alarm!';
// definice zdroje S
source S = 'signal', name, 'value quality';

// zvoleni implicitniho zdroje
def_source = A;


// konkretni dopoctove rovnice
```

```
// pokud neni kvalita veliciny gAGC:SL:P aktualni
// pri neaktualnosti kvality veliciny gAGC:SLM:P
// neuvazovat jeji hodnotu
gAGC:SL:t_st = iif( !qvalid(gAGC:SL:P), _I(gAGC:SLM:P) );

// pokud ma meric poruchu -> v saldu neuvazovat
gALB4:V443:P_kz = iif( S:gALB4:V443:s_SRC == 0, 0,
    // pokud predavaci bod nelezi na uzemi CR -> v saldu neuvazovat
    iif( switch( S:gALB4:V443:s_SRC,
        1, S:MER_CR,
        2, S:MER_CR,
        3, S:MER_ZAHR ) == S:MER_ZAHR, 0,
        // pokud jde o import energie -> ztraty odecist
        // jinak ztraty pricist
        iif( S:MER_CR, -ztraty(A:gALB4:V443:s_SRC, 4.67, 595, 415),
            ztraty(A:gALB4:V443:s_SRC, 2.53, 418, 415)
        )
    )
);
```

Dodatek C

Obsah DVD

jméno	**popis**
`text/`	
`projekt.pdf`	elektronická verze textu bakalářské práce ve formátu PDF
`src/`	zdrojové soubory textu bakalářské práce v jazyce LaTeX
`src/`	zdrojové soubory překladače a interpretu
`doc/`	
`user/`	uživatelská dokumentace jazyka dopočtových rovnic ve formátu PDF
`prog/`	programátorská dokumentace zdrojových souborů ve formátu HTML
`bin/`	
`ris.dvi`	obraz systému Debian s instalací systému *Ris* pro VirtualBox
`README`	popis adresářové struktury DVD a návod k použití